LIBRARY OF THE
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

510.84
Il6r
no.433-438
cop.2

Digitized by the Internet Archive
in 2013

http://archive.org/details/illiaciiireferen434mcco

510.84
Il6r
no.434

Math

Report No. 434

COO-2118-0006

ILLIAC III REFERENCE MANUAL

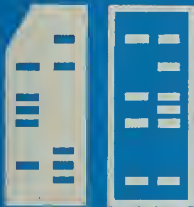
VOLUME II: Instruction Repertoire

edited by
B. H. McCormick and B. J. Nordmann, Jr.

D. E. Atkins, R. T. Borovec, L. N. Goyal, L. M. Katoh,
R. M. Lansford, J. C. Schwebel and V. G. Tareski

THE LIBRARY OF THE
MAY 18 1971
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

February 26, 1971



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

COO-2118-0006

Report No. 434

ILLIAC III REFERENCE MANUAL

VOLUME II: Instruction Repertoire

edited by

B. H. McCormick and B. J. Nordmann, Jr.

D. E. Atkins, R. T. Borovec, L. N. Goyal, L. M. Katoh,
R. M. Lansford, J. C. Schwebel and V. G. Tareski

February 26, 1971

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

This work was supported by Contract AT(11-1)-1018 with the U.S. Atomic Energy Commission through September 30, 1970. Current support is under Contract AT(11-1)-2118 with the above agency.

ABSTRACT

The Illiac III Reference Manual is being issued in this final documentation as four volumes:

Volume I;	The Computer System
This issue---Volume II:	Instruction Repertoire
Volume III;	Input/Output
Volume IV;	Supervisor Organization

For ease of cross-reference an integrated table of contents will be issued separately.

This section (Volume II) deals with the extraordinarily diverse instruction repertoire of the Illinois Pattern Recognition Computer (Illiac III). As the "central processing units" of the machine, a Taxicrinic Processor interprets these instructions and, if appropriate, routes the relevant operands and control information to units of the machine for execution: to the main store, arithmetic unit, pattern articulation unit, and interrupt unit. Indirectly, by mediation of the interrupt unit, the Taxicrinic Processor can also initiate and terminate I/O operations.

Certain novel aspects of the instruction repertoire are reiterated here. First, the Taxicrinic Processors interpret the unique instructions to control plane parallel picture processing in the Pattern Articulation Unit. (See Section 2.4) Secondly, because the latter phases of image analysis so commonly involves graph transformations, the machine code emphasizes a set of list processing instructions. Again to permit ready implementation of list processing and graph transformation languages, the imprimitive instructions are introduced in Section 2.2.10. These provide the basis for a hardware-implemented macro-assembler, and allow for efficient use of interpretive realizations of programming languages.

Finally it should be noted that the instruction repertoire, particularly in the so-called "system instructions", Section 2.2.11, takes full cognizance of the special problems facing an operating system coordinating multiple processors in a multi-programming environment.

ACKNOWLEDGMENTS

The Illiac III Reference Manual is based in part upon two earlier reports:

(1) B.H. McCormick (editor), William D. Bond, Kimio Ibuki, Roger E. Wiegel and John A. Wilber, Preliminary Programming Manual for the Illiac III Computer, Department of Computer Science Report 185, University of Illinois, July 1965.

(2) B.H. McCormick and R.M. Lansford (editors), D.E. Atkins, R.T. Borovec, G.N. Cederquist, S.K.Chan, L.A. Dunn, J.P. Hayes, L.M. Katoh, P.L. Koo, B.J. Nordmann, Jr., J.A. Rohr, and J.C. Schwebel, Illiac III Programming Manual, Department of Computer Science Manual, University of Illinois, March 1968.

The present editors (B.H. McCormick and B.J. Nordmann, Jr.) acknowledge with gratitude the contribution of these earlier groups as transplanted to this greatly expanded manual. In addition, Dr. Rangaswamy Narasimhan contributed significantly to the early definition of the pattern articulation unit instruction set. In like manner Philip Merryman assisted materially in the early formulation of machine-implemented macros--the imprimitive instructions of Illiac III.

Authors and contributors to the manual are listed by principal area of concern:

Taxicrinic Processor:	B.J. Nordmann, Jr., R.M. Lansford, J.C. Schwebel, R.E. Wiegel
Input/Output Processor:	L.M. Katoh, V.G. Tareski, J.V. Wenta, G.N. Cederquist, J.P. Hayes
Arithmetic Units:	D.E. Atkins, L.M. Goyal
Pattern Articulation Unit:	R.T. Borovec, R.P. Harms, G.T. Lewis
Interrupt Unit:	L.M. Goyal
Exchange Net:	S.K. Chan, P. Krabbe
Scanner-Monitor-Video	L.A. Dunn, L.M. Goyal, V.G. Tareski, R.G. Martin, R.C. Amendola
Supervisor Organization:	B.J. Nordmann, Jr., R. M. Lansford

The seemingly endless drafts and revisions of this manual have been handled with patience and fortitude by Mrs. Betty Gunsalus and Mrs. Roberta Andre'. Illustrations were prepared by Mr. Stanley Zundo.

2. INSTRUCTION REPERTOIRE

- 2.1 Instruction Format
  - 2.1.1 Mnemonic Byte
  - 2.1.2 Operand Phrases
    - 2.1.2.1 Tag Field (TAG)
    - 2.1.2.2 Slash Operation Field (SØ)
    - 2.1.2.3 Indirect Bit (I)
    - 2.1.2.4 Last Bit (L)
    - 2.1.2.5 Flag Bit (F)
    - 2.1.2.6 Modifier Field (M_0, M_1)
    - 2.1.2.7 Modification Operation Bits (m_0, m_1)
    - 2.1.2.8 Sequence of Operations for Operand Phrases
- 2.2 Instructions Executed by the Taxicrinic Processor
  - 2.2.1 Single Cycle Data Transfers
    - 2.2.1.1 Assign
    - 2.2.1.2 Operand Stack Instructions
      - 2.2.1.2.1 Push
      - 2.2.1.2.2 Pop
      - 2.2.1.2.3 Load
      - 2.2.1.2.4 Store
    - 2.2.1.3 Masked Operations
      - 2.2.1.3.1 Set
      - 2.2.1.3.2 Reset
      - 2.2.1.3.3 Test
      - 2.2.1.3.4 Test with Mask
  - 2.2.2 Multiple Cycle Data Transfers
    - 2.2.2.1 Operand Stack Field Transfers
      - 2.2.2.1.1 Push Field
      - 2.2.2.1.2 Push Field Reverse
      - 2.2.2.1.3 Pop Field
      - 2.2.2.1.4 Pop Field Reverse
      - 2.2.2.1.5 Pointer Manipulation
    - 2.2.2.2 Field Comparison
      - 2.2.2.2.1 Scan
      - 2.2.2.2.2 Scan with Mask
    - 2.2.2.3 Decimal Conversion
      - 2.2.2.3.1 Pack Numeric
      - 2.2.2.3.2 Unpack Numeric

- 2.2.2.4 String Manipulation
  - 2.2.2.4.1 Move
  - 2.2.2.4.2 Translate
  - 2.2.2.4.3 Edit
- 2.2.3 Stack Utility
  - 2.2.3.1 Exchange
  - 2.2.3.2 Duplicate
  - 2.2.3.3 Sluff
- 2.2.4 Logical Operations
  - 2.2.4.1 Unary Logical
    - 2.2.4.1.1 Zero
    - 2.2.4.1.2 One
    - 2.2.4.1.3 Not
    - 2.2.4.1.4 Count Ønes
    - 2.2.4.1.5 Bit
  - 2.2.4.2 Binary Logical
    - 2.2.4.2.1 And
    - 2.2.4.2.2 Or
    - 2.2.4.2.3 Exclusive Or
    - 2.2.4.2.4 Equivalence
    - 2.2.4.2.5 Compare Logically
- 2.2.5 Shift
  - 2.2.5.1 Left Shift
  - 2.2.5.2 Right Shift
- 2.2.6 Conditional Instructions
  - 2.2.6.1 If
  - 2.2.6.2 If Not
- 2.2.7 Utility
  - 2.2.7.1 Location
  - 2.2.7.2 Specify
  - 2.2.7.3 No Operation
- 2.2.8 Arithmetic Operations
  - 2.2.8.1 Unary Arithmetic
    - 2.2.8.1.1 Negate
    - 2.2.8.1.2 Absolute Value
    - 2.2.8.1.3 Minus
    - 2.2.8.1.4 Test Algebraically
    - 2.2.8.1.5 Convert Short to Long Fixed Point

2.2.8.2 Binary Arithmetic
2.2.8.2.1 Fixed Point Addition
2.2.8.2.2 Fixed Point Subtraction
2.2.8.2.3 Fixed Point Algebraic Comparison
2.2.9 List Processing Instructions
2.2.9.1 Sequence Left
2.2.9.2 Sequence Right
2.2.9.3 Get Cell
2.2.9.4 Put Cell
2.2.9.5 Insert Cell Left
2.2.9.6 Insert Cell Right
2.2.9.7 Delete Cell Left
2.2.9.8 Delete Cell Right
2.2.10 Imprimitive Instructions
2.2.10.1 Operational Description
2.2.10.2 Go To
2.2.10.3 Execute
2.2.10.4 Call
2.2.10.5 Exit
2.2.10.6 Name Permutation and Repermutation
2.2.11 System Instructions
2.2.11.1 Supervisor Operation
2.2.11.1.1 Supervisor Call
2.2.11.1.2 Supervisor Return
2.2.11.1.3 Rename
2.2.11.1.4 Sleep
2.2.11.1.5 Activate TP
2.2.11.1.6 Reserve Unit
2.2.11.1.7 Load Task Register
2.2.11.1.8 Store Task Registers
2.2.11.2 Interrupt Handling
2.2.11.2.1 Set Interrupt Mask
2.2.11.2.2 Interrupt Return

2.2.11.3 Input/Output
2.2.11.3.1 Start I/O
2.2.11.3.2 Halt I/O
2.2.11.3.3 Load IOP Base Register
2.2.11.4 Coordination
2.2.11.4.1 Increment and Check
2.2.11.4.2 Link
2.2.11.4.3 Who
2.2.11.5 Timing
2.2.11.5.1 Read Clock
2.2.11.5.2 Set Timer
2.2.11.5.3 Read Timer
2.3 Instructions Executed by Arithmetic Units
2.3.1 Arithmetic Data Formats
2.3.1.1 Short Fixed Point
2.3.1.2 Long Fixed Point
2.3.1.3 Floating Point
2.3.1.4 Decimal
2.3.2 Arithmetic Instructions
2.3.2.1 Add
2.3.2.2 Subtract
2.3.2.3 Multiply
2.3.2.4 Divide
2.3.2.5 Compare Algebraically
2.3.2.6 Convert to Decimal
2.3.2.7 Convert to Floating Point
2.3.2.8 Convert to Long Fixed Point
2.3.2.9 Polynomial Evaluation
2.3.3 Exceptional Conditions for Arithmetic Instructions
2.3.3.1 General
2.3.3.2 Overflow (OV)
2.3.3.3 Underflow (UN)
2.3.3.4 Invalid Decimal Data (ID)
2.3.3.5 Loss of Significance (LS)

2.4 Instructions Executed by the Pattern Articulation Unit
- 2.4.1 Conventions
  - 2.4.1.1 Planes and Borders
  - 2.4.1.2 Direction Numbers
  - 2.4.1.3 Indicators
    - 2.4.1.3.1 Exceptional Conditions
    - 2.4.1.3.2 Indicator Halfword
- 2.4.2 Zero-Plane Instructions
  - 2.4.2.1 Topology
  - 2.4.2.2 Set Origin
  - 2.4.2.3 Resume
  - 2.4.2.4 Restart
- 2.4.3 One-Plane Instructions
  - 2.4.3.1 Data Formats
    - 2.4.3.1.1 Coordinate Mode
    - 2.4.3.1.2 Incremental Code
  - 2.4.3.2 Clearp
  - 2.4.3.3 Setp
  - 2.4.3.4 Testp
  - 2.4.3.5 Testb
  - 2.4.3.6 Replicate
  - 2.4.3.7 Shift
  - 2.4.3.8 Tally
  - 2.4.3.9 Tallyho
  - 2.4.3.10 Area
  - 2.4.3.11 List
  - 2.4.3.12 Listsz
  - 2.4.3.13 Listlz
  - 2.4.3.14 Listi
  - 2.4.3.15 Readlz
  - 2.4.3.16 Rderlz
  - 2.4.3.17 Erasep
  - 2.4.3.18 Plot
  - 2.4.3.19 Plotsz
  - 2.4.3.20 Plotlz

2.4.3.21 Ploti
2.4.3.22 Writlz
2.4.3.23 Wrerlz
2.4.4 Two-Plane Instructions
2.4.4.1 Copy
2.4.4.2 Copyc
2.4.4.3 Pland
2.4.4.4 Plor
2.4.4.5 Plnand
2.4.4.6 Plnor
2.4.4.7 Plexor
2.4.4.8 Pleqv
2.4.5 Three-Plane Instructions
2.4.5.1 Connect
2.4.6 Multiple-Plane Instructions
2.4.6.1 Boole
2.4.6.2 Gate IA
2.4.7 Border Instructions
2.4.7.1 Data Formats
2.4.7.1.1 Raster String
2.4.7.1.2 GBW Byte
2.4.7.2 Loadb
2.4.7.3 Storeb
2.4.7.4 Pushb
2.4.7.5 Popb
2.4.7.6 Moveb
2.4.8 Information on Boole Instruction
2.4.8.1 Availability List
2.4.8.2 Form for Specifying General Boolean Functions
2.4.8.3 Examples Illustrating the Coding
2.4.8.3.1 Purely Horizontal Logic
2.4.8.3.2 Purely Vertical Logic
2.4.8.3.3 General Three-Dimensional Logic

2. INSTRUCTION REPERTOIRE

This section (Volume II) deals with the extraordinarily diverse instruction repertoire of the Illinois Pattern Recognition Computer (Illiac III). As the "central processing units" of the machine, a Taxicrinic Processor interprets these instructions and, if appropriate, routes the relevant operands and control information to units of the machine for execution: to the main store, arithmetic unit, pattern articulation unit, and interrupt unit. Indirectly, by mediation of the interrupt unit, the Taxicrinic Processor can also initiate and terminate I/O operations.

Certain novel aspects of the instruction repertoire are reiterated here. First, the Taxicrinic Processors interpret the unique instructions to control plane parallel picture processing in the Pattern Articulation Unit. (See Section 2.4) Secondly, because the latter phases of image analysis so commonly involves graph transformations, the machine code emphasizes a set of list processing instructions. Again to permit ready implementation of list processing and graph transformation languages, the imprimitive instructions are introduced in Section 2.2.10. These provide the basis for a hardware-implemented macro-assembler, and allow for efficient use of interpretive realizations of programming languages.

Finally it should be noted that the instruction repertoire, particularly in the so-called "system instructions", Section 2.2.11, takes full cognizance of the special problems facing an operating system coordinating multiple processors in a multi-programming environment.

2.1 Instruction Format

Every Illiac III instruction can be considered to be in prefix form: an operation (specified by a mnemonic byte)followed by operands (designated by operand phrases), if any. An instruction with n operands in main storage has n operand phrases respectively, each one implicitly specifying the data address of the operand. In addition, an instruction may call upon operands from the Operand Stack. In this case, the operand address is implied by the mnemonic byte.

The general format for the Illiac III instructions is shown in figure 2.1.

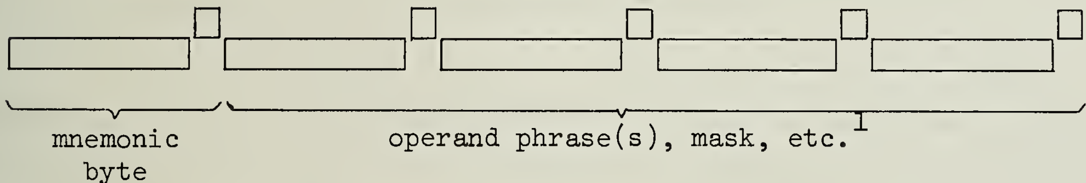


Figure 2.1 Illiac III Instruction Format

This format consists of a single mnemonic byte normally followed by one or more operand phrases. These phrases may be long or short(see Section 2.1.2).

In primitive instructions the total length of these operand phrases may not exceed 4 bytes. Some few primitive instructions, while adhering to the 4-byte constraint, have fields with alternate interpretation: mask, etc.

In imprimitive instructions there may be up to 12 operand phrases. Here however, no restriction is placed upon the total length of the instruction.

---

1 For imprimitive instructions up to 12 operand phrases are allowed (36 bytes max.).

2.1.1 Mnemonic Byte

Every instruction is defined and initiated by a single mnemonic byte. The mnemonic byte may alone define the instruction or it may have associated with it one or more operand phrases and/or an auxiliary condition mask.

The instructions are partitioned into eight classes by three fixed bits of the mnemonic byte (specifically, by the first 2 prefix bits and the flag bit). Some of these classes correspond to familiar programming instruction sets: arithmetic, pattern articulation, etc. Others designate the number of associated operands: zero, one, two (for primitives) or arbitrary (for imprimatives).

The two low order data bits of the mnemonic byte are often reserved for the Field Designator (FD) or the Number Type (NT). This latter situation is peculiar to arithmetic instructions and is discussed separately in Section 2.3.

The Field Designator (FD) is the operand field length (1, 2, 4 or 8 bytes) if the operand phrase(s) refer to a core address. If the operand phrase(s) refer to a Pointer Register, these bits indicate the pointer field to be used, i.e. Link, Value, Segment Name or appropriate combination thereof.

If the Immediate option was not specified by the (either) operand phrase, the fields designated are used to refer to a core or stack field; all flags of each field are transmitted.

00	Byte
01	Halfword
10	Word (4 bytes)
11	Double word (8 bytes)

If the (either) operand phrase specifies the Immediate option, the fields then designated are used to refer to program register fields.

00	Value (no flags, halfword)
01	Link (no flags, halfword)
10	Link and Value (all flags, word)
11	Segment Name (no flags, halfword)

Not all instructions permit the Immediate option; see instruction text where in doubt.

2.1.2 Operand Phrases

Operand phrases provide a uniform technique throughout Illiac III for addressing main storage and for operating on the 15 pointer stacks. In an operand phrase, the file (main store) address of an operand is implied by giving the name of its associated pointer stack. That is, the data address is specified by the topmost pointer in the pointer stack named by the operand phrase tag field.

In addition to naming a pointer stack, an operand phrase may also specify operations which modify the value of the pointer and/or change the depth of the pointer stack. These operations take place before, after, or both before and after the actual execution of the kernel instruction.

The length of each operand phrase (1 or 3 bytes) is specified by the flag of the first byte of each phrase: a flag of '0' indicates a short phrase; a flag of '1' indicates a long phrase.

> For primitive instructions a maximum of two operand phrases (only one of which can be long) is allowed. Additional single byte fields may be intermingled if prescribed by the instruction format.
>
> For imprimitive instructions a maximum of 14 operand phrases (long or short) is permitted. No other fields may be intermingled. The terminal phrase is specified by the low order data bit of the first byte of the operand phrase: '0' if more phrases follow, '1' for the terminal phrase.

Accordingly the instruction field is consecutively partitioned into phrases until (a) the number of phrases specified by the mnemonic byte is exhausted (**primitives**), or (b) a terminal bit is sensed (imprimitives).

Every operand, other than those in the top of the Operand Stack, is designated by the use of an Operand Phrase.

Every operand phrase has one of two forms: a single-byte short phrase or a three-byte long phrase. A short phrase is indicated when the flag bit of the first byte is '0'; a long phrase is indicated when this bit is '1'.

The fields of the two types of operand phrases are shown in Figure 2.1.2.

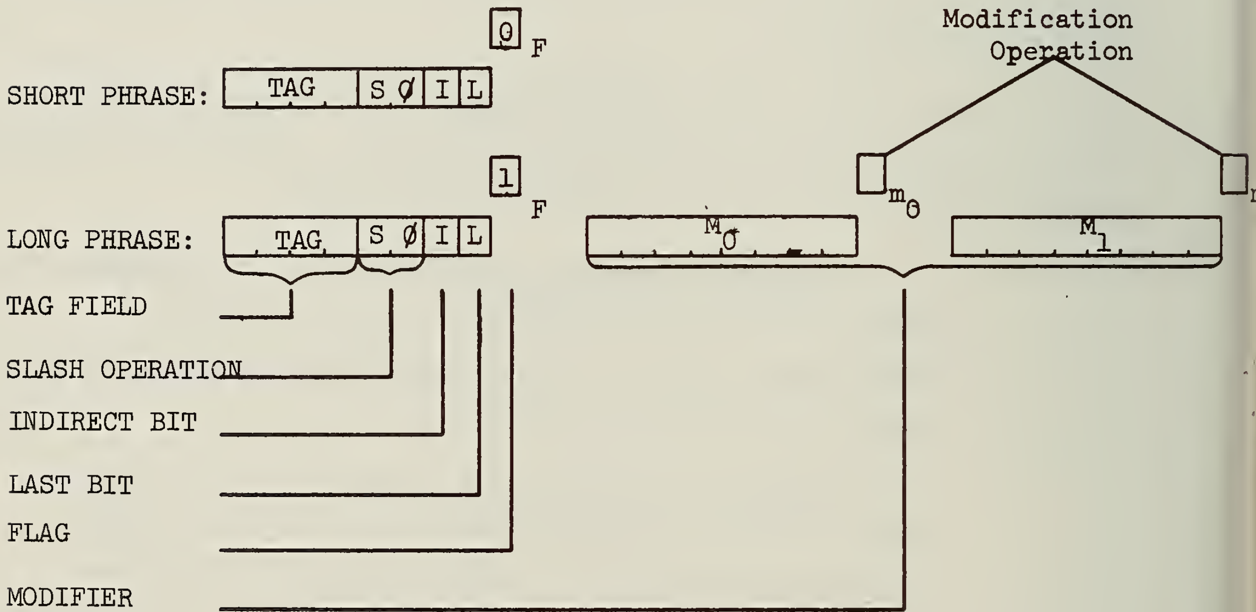


Figure 2.1.2 Operand Phrase Format

2.1.2.1 Tag Field (TAG)

The first 4 bits of the first byte of the operand phrase is the tag field. This field designates, or names, one of the 15 pointer registers which is to be operated, on renamed, or otherwise employed in the construction of the operand address.

In a Taxicrinic Processor there are 15 pointer registers, each having an associated 4-bit name register. As implied, the name register holds the current name (e.g., PR_3) of the pointer register. At any given time each pointer register must have a unique name, i.e., no two name registers may contain the same name.

When an imprimitive instruction is executed, some or all of the names of the pointer registers may be permuted, i.e., some or all of the name registers may be changed, but after name permutation each pointer register will still have a unique name. For the CALL and EXECUTE imprimitive instructions the new name is specified by the position of the phrase and its tag: the tag of the operator phrase specifies which pointer register may be renamed PR_0; the tag of the next operand phrase specifies which pointer register may be renamed PR_1; the tag of the third operand phrase specifies which pointer register may be renamed PR_2; etc. If there are less than 15 operand phrases the remaining pointer register names are automatically permuted as described below in Section 2.2.10.6.

For the CALL and EXECUTE imprimitive instructions, name changing consists of possibly permuting every pointer register name. For the GØTØ imprimitive instruction, name changing consists only of possibly swapping the names of PR_0 and the pointer register designated by the tag field of the operator phrase. For any imprimitive instruction the appropriate type of name change (permutation or swapping) will occur unless a conditional subtraction failure occurs for some modification as described below in Section 2.2.10.3.

For imprimitive instructions two special conventions are adhered to:

Because of its importance for imprimitive instructions, the first such operand phrase, should it exist, will be designated as the operator phrase. The function of the operator phrase is to control the instruction pointer, PR_0, and thus control program flow. The format of the operator phrase is the same as for a general operand phrase, except for a minor restriction: the SØ field is ignored.

In every imprimitive operand phrase, the last data bit of the first byte is a Last bit (L). If L = '0', more operand phrases follow; if L = '1', this is the terminal (i.e., last) operand phrase of the instruction.

For imprimitive instructions only, then, the tag field of the operator phrase specifies which pointer register may be renamed PR_0. If the tag of the operator phrase is 0, the current instruction pointer register is modified as specified by the modifier of the phrase. If the tag of the operator phrase is not zero, the specified pointer is modified by the modifier of the operator phrase, and then that pointer may be renamed PR_0.

2.1.2.2 Slash Operation Field (SØ)

The slash operations control the "pushing" and "popping" of the designated pointer stack. The first bit of the slash operation, $SØ_0$ is called the pre-slash and, if set to '1', indicates that the pointer stack is to be duplicated (pushed) before executing the instruction. Bit $SØ_1$ is called the post-slash, and if set to '1', indicates that the pointer stack is to be popped after execution of the instruction. The following table gives the interpretation of the four possible variants:

Pre-Slash Bit	Post-Slash Bit	Operand Operation	Initial/Final Operation	
0	0	FINAL VALUE	no push	no pop
0	1	POP ON EXIT	no push	pop
1	0	SAVE INITIAL VALUE	push	no pop
1	1	INITIAL VALUE	push	pop

The interpretation expresses the status of the pointer of the named operand file upon completion of the instruction.

Note: For imprimitive instructions the slash operations specified for an operator phrase will be ignored; they are implied by the nmemonic byte.

2.1.2.3 Indirect Bit (I)

The interpretation of this bit depends on whether the phrase is short or long. If the phrase is short, this bit is used to specify Indirect addressing. If the phrase is long, this bit is used to specify an Indirect Modifier.

Indirect Addressing: If the phrase is short, the indirect bit designates whether the effective operand address is computed directly (I=0) or indirectly (I=1) from the specified pointer register. If indirect addressing is specified, the pointer register specified by the tag of the phrase is used to access a halfword in core (flag bits excluded). This halfword then replaces the value of the pointer register specified by the short phrase.

Indirect Modifier: If the phrase is long, this bit designates whether the Modifier Field (M_0, M_1) contains the modifier itself (I=0) or whether the Modifier points to a pointer register (or ØS) which contains the modifier (I=1).

2.1.2.4 Last Bit (L)

The last data bit of the first byte of an instruction operand phrase has one of four interpretations depending on the instruction.

Last Bit (L): In every imprimitive operand phrase, if L = '0', more operand phrases follow; if L = '1', this is the terminal (i.e., last) operand phrase of the instruction.

Count Phrase Bit (Ct): If the primitive instruction requires a count, e.g. Push Field, then the first operand phrase is a count phrase. If Ct = '1', then the pointer value specified by the tag is used as the count; if Ct = '0', no count exists and a flag on the data is used to terminate the instruction.

Immediate Operand Bit (Imm): Some primitive instructions allow the possibility of using a pointer register (or some portion of it) as an immediate operand rather than as the address of an operand. If Imm = '0', the pointer will be used as an address. For those instructions which allow Immediate operands, if Imm = '1', the Field Designator bits (FD) will be interpreted as follows:

00 Value field: right half of the PR, no flags
01 Link field: left half of the PR, no flags
10 Register: all of the PR, including flags
11 Segment Name field: Segment Name Register corresponding to PR, no flags.

The instruction then applies to these fields of the pointer specified by the tag of the phrase.

Note: Operand phrases which may have immediate operands (count) are marked in this manual with an Imm (Ct) as a superscript immediately following the phrase.

Unused: For many primitive instructions the last data bit of the first byte of an operand is unused.

2.1.2.5 Flag Bit (F)

The flag bit always indicates whether the phrase is long or short. For all operator/operand phrases if the flag bit is '0', the phrase is short; if the flag bit is '1' the phrase is long and the next two bytes specify a modifier to be applied to the pointer register (value).

2.1.2.6 Modifier Field (M_0, M_1)

A modifier field occurs only for a long phrase and consists of the data bits of the second and third bytes of the phrase denoted by M_0 and M_1.

If the Indirect bit (I) = '0', the modifier is used directly as specified by the Modification Operation Bits m_0 and m_1. The modifier value M is considered to be a 16-bit positive integer.

If the Indirect bit = '1', the leftmost four bits of M_0 are used to specify a Secondary Tag. The pointer value of the specified secondary register is applied to the primary register as specified by the Modification Operation Bits m_0 and m_1. The secondary tag may specify PR_0, ..., PR_{15}. PR_{15} is interpreted to mean that the Indirect Modifier is to be taken from the top halfword cell in the ØS. The contents of the secondary register (or ØS) are not changed by this process.

2.1.2.7 Modification Operation Bits (m_0, m_1)

The modification operation bits, (the flag bits on the second and third bytes of a long operand phrase) denoted m_0 and m_1, are used to specify the operation to be performed on the pointer value after the pre-slash operation (if any). The four possible operations are specified in the table below.

m_0	m_1	Operation	Symbol
0	0	REPLACEMENT	=
0	1	ADDITION	+
1	0	CONDITIONAL SUBTRACTION	-
1	1	NOT USED	

REPLACEMENT causes the modifier M(or if Indirectly Modified, the Secondary Pointer value) to replace the pointer value specified by the tag of the phrase before execution of the instruction.

ADDITION causes the modifier M (or if Indirectly Modified, the Secondary Pointer value) to be added, modulo 2^{16} to the pointer value specified by the tag of the phrase before execution of the instruction.

Note: If the tag of a phrase is zero, the addition is done on the address of the first byte of the instruction; and a transfer of control will occur prior to the execution of the next instruction.

CONDITIONAL SUBTRACTION causes calculation of a test value by subtracting the modifier M (or if Indirectly Modified, the Secondary Pointer value) from the pointer value specified by the tag (both treated as positive 16-bit integers) before execution of the instruction.

If the test value is greater than zero, the test value replaces the specified pointer value and instruction processing continues. If the test value is less than or equal to zero, the specified pointer value is not modified. Then after all operand phrases have been processed, if a conditional subtraction has been attempted, the Conditional Subtraction indicator (CS) is reset to the outcome of the 'OR' of any (possibly multiple) attempted conditional subtractions.

If any conditional subtraction has failed, the instruction is not executed, and any modifications performed on PR_0 are cancelled. Execution continues with the next instruction in sequence.

2.1.2.8 Sequence of Operations for Operand Phrases

The following sequence of operations is used to process operand phrases:

I. For the operator phrase of any imprimitive instruction:

1. If REPLACEMENT, ADDITION, or CONDITIONAL SUBTRACTION is specified, it is performed on the specified pointer value.

Note: Slashing operations specified for an operator phrase will be ignored.

II. For each operand phrase:

1. Preslash. If the preslash bit is '1', a copy of the specified pointer is pushed into its associated pointer stack.

2.1 Short Phrase. If the Indirect addressing bit is '1', the specified pointer value is modified (REPLACEMENT) by the halfword modifier in core.

2.2 Long Phrase. If REPLACEMENT, ADDITION, or CONDITIONAL SUBTRACTION is specified, it is performed on the specified pointer value.

III. For each instruction (following the prescan):

1.1 If CONDITIONAL SUBTRACTION was not attempted, the instruction is executed.

1.2 If CONDITIONAL SUBTRACTION was attempted, the CS indicator is reset to the 'OR' of the results of the individually attempted CS's.

1.2.1 If no CS failed, the instruction is executed.

1.2.2 If any phrase failed CS, no name changing or transfer of control occurs and any modifications performed on PR_0 are cancelled. For CALL and EXECUTE processing of the imprimitive continues with the past-scan.

IV. For each operand phrase:

1. Postslash. If the postslash bit is '1', the pointer at the top of the specified stack is popped out.

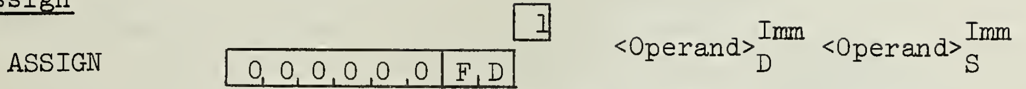
V. After all operand phrases have been considered for post-operations, control is passed to the location designated by PR_0. If there was a CONDITIONAL SUBTRACTION failure, the new location will be that of the next instruction in core.

2.2 Instructions Executed by the Taxicrinic Processors

By definition all Illiac III instructions are initially interpreted by a Taxicrinic Processor. The instructions described in this section are, however, restricted to those whose execution is performed entirely by a TP. This set includes instructions for data transfer, operand stack modification, logical/shift operations, fixed point addition/subtraction and assorted supervisor actions.

2.2.1 Single Cycle Data Transfers

2.2.1.1 Assign



This two operand instruction may be used to transfer data of variable field length to the address specified by the 'Destination' phrase (to the pointer itself if the Immediate option was specified) from the address specified by the 'Source' phrase (from the pointer register itself if the Immediate option has been specified.)

NOTE: Since the length of the instruction is restricted to at most 5 bytes, both operand phrases cannot be long. However, either or both may be short.

By use of the Immediate option, the following transfers may be obtained:

Destination \ Source	Immediate = '0'	Immediate = '1'
Immediate = '0'	To Core From Core	To Core From Register
Immediate = '1'	To Register From Core	To Register From Register

Indicators: bounds overflow, parity check.

2.2.1.2 Operand Stack Instructions

This set of (4) single operand instructions provides for moving data between the top of the OS and core memory (or pointer registers, if the Immediate option is specified in the operand phrase). For these instructions, operand address will designate either a core address (Imm = 0) or a program register (Imm = 1).

2.2.1.2.1 Push

PUSH [0 1 0 0 0 0 F D] [0] <Operand>$^{Imm}_{S}$

A single operand field is transferred from the operand address to the OS. The Operand Stack Pointer (OSP-PR#13) is incremented by FD.

Indicators: bounds overflow, parity check.

2.2.1.2.2 Pop

POP [0 1 0 0 1 0 F D] [0] <Operand>$^{Imm}_{D}$

A single operand field is transferred from the OS to the operand address. The ØSP is decremented by FD.

Indicators: bounds overflow, parity check.

2.2.1.2.3 Load

LD [0 1 0 0 1 0 F D] [0] <Operand>$^{Imm}_{S}$

The OSP is decremented by FD. A 'PUSH' instruction is then executed. (Effectively, the topmost field of size FD in the OS is overwritten by the field at the operand address.)

Indicators: bounds overflow, parity check.

2.2.1.2.4 Store

ST [0 1 0 0 1 1 F D] [0] <Operand>$^{Imm}_{D}$

A 'POP' instruction is executed; the OSP is then incremented by FD. The OS is unchanged. (Effectively, the top FD cell in the OS is duplicated at the operand address.)

Indicators: bounds overflow, parity check.

2.2.1.3 Masked Operations

2.2.1.3.1 Set

SET [0 | 1 | 0 | 1 | 0 | 0 | F | D] [0]   <Operand>$^{Imm}_{D}$

The topmost FD field in the OS is treated as a mask. Each bit of the mask which is '1' defines a corresponding bit in the field at the operand address which is to be set to '1'. Bits in positions of the operand field which correspond to zeros in the mask are unchanged. The OS is not changed. (Effectively, an 'OR' between the mask and operand field is performed.)

Indicators: bounds overflow, parity check.

2.2.1.3.2 Reset

RESET [0 | 1 | 0 | 1 | 0 | 1 | F | D] [0]   <Operand>$^{Imm}_{D}$

The topmost FD field in the OS is treated as a mask. Each bit of the mask which is '1' defines a corresponding bit in the field at the operand address which is to be set to zero. Bits in positions of the operand field which correspond to zeros in the mask remain unchanged. The OS is not changed. (Effectively, the mask is complemented and AND'ed with the operand field.)

Indicators: bounds overflow, parity check.

2.2.1.3.3 Test

TEST [0 | 1 | 0 | 1 | 1 | 0 | F | D] [0]   <Operand>$^{Imm}_{D}$

The topmost FD cell in the ØS is treated as a bit pattern. Bits in the top of the ØS are logically compared with the bit pattern of the operand. The ØS is unchanged. Status indicators are set according to the comparison results.

Indicators: bounds overflow, parity check, greater, equal, less, flags match.

2.2.1.3.4 Test with Mask [O] <Operand>$^{Imm}_{D}$

TESTM | 0 | 1 | 0 | 1 | 1 | 1 | F | D |

The topmost FD field in the OS is treated as a bit pattern. The next-to-top FD field in the OS is treated as a mask.

Bits of the mask which are '1' define corresponding bits in the cell at top of the ØS which are to be logically compared with the bit pattern of the source operand. The defined data bits are compared for match or no match. The OS is unchanged. (Effectively, the mask is AND'ed with the data cell which is then logically compared with the bit pattern.) Status indicators are set according to the comparison results.

Indicators: bounds overflow, parity check, greater, equal, less, flags match.

2.2.2 Multiple Cycle Data Transfers

This class of (11) instructions facilitates operations on relatively long data fields. To accomplish this, several cycles of a simpler type are performed.

All multiple cycle instructions require a count (Ct) phrase and one or two operand phrases. If the count bit (Section 2.1.2.4) is on (Ct = '1'), the specified pointer register contains a count, denoted M. The instruction will operate on M cells of data. If the count bit of a count phrase is off (Ct = '0'), the instruction continues until a byte of the source data has a flag set to '1'. The cell with the flag set to '1' is the last cell processed. If the cell size is not a byte, the rightmost flag in the cell set to '1' will terminate the instruction.

2.2.2.1 Operand Stack Field Transfers

This set of 4 instructions may be used to transfer variable length fields to and from the Operand Stack.

Programming note: Boundary alignment (or the lack of it) may seriously affect the execution speed of these instructions. Namely, if the specified field does not start on a double word boundary, the instruction will proceed a cell at a time until a double word boundary is encountered. Execution then proceeds a double word at a time. Depending on the alignment of the end of the field, the instruction may revert to a cell at a time for the final fraction of a double word.

2.2.2.1.1 Push Field [1]

PUSHF [0 0 1 0 0 0 | F D] <Count> <Operand>_S

The field beginning at the address specified by the source operand is pushed into the Operand Stack. The order of the cells is preserved. See Section 2.2.2.1.5. Following

execution, the source Pointer Register will contain the address of the cell following the last one processed.

Indicators: bounds overflow, parity check

2.2.2.1.2 Push Field Reverse [1]

PUSHFR | 0 0 1 0 0 1 | F D | <Count> <Operand>$_S$

The field ending at the address specified by the source phrase is pushed into the ØS. The order of the cells is reversed. Following execution the source Pointer Register will contain the address of the last cell processed. See Section 2.2.2.1.5.

Indicators: bounds overflow, parity check

2.2.2.1.3 Pop Field [1]

PØPF | 0 0 0 0 0 1 | F D | <Count> <Operand>$_D$

Data are popped from the Operand Stack to the location specified by the destination operand. The order of the cells is preserved. Following execution the destination Pointer Register will contain the address of the last cell processed. (The pointer is decremented during execution, see Section 2.2.2.1.5.)

Indicators: bounds overflow, parity check

2.2.2.1.4 Pop Field Reverse [1]

PØPFR | 0 0 0 0 1 0 | F D | <Count> <Operand>$_D$

Data is popped from the Operand Stack to the location.

specified by the destination phrase. The order of the cells is reversed. Following execution, the destination Pointer Register will contain the address of the cell following the last cell processed. See Section 2.2.2.1.5.

Indicators: bounds overflow, parity check

2.2.2.1.5 Pointer Manipulation

The movement of the data pointers during execution of instructions 2.2.2.1.1 thru 2.2.2.1.4 is summarized on the following page.

For each instruction, two data fields (byte length cells) are represented: one in core, the second in the Operand Stack.

The symbols ↓ , ↓̸ are used to represent the initial and final positions of the respective pointers (core, ØS).

The dotted arrow to the right of each figure indicates the direction of the data flow.

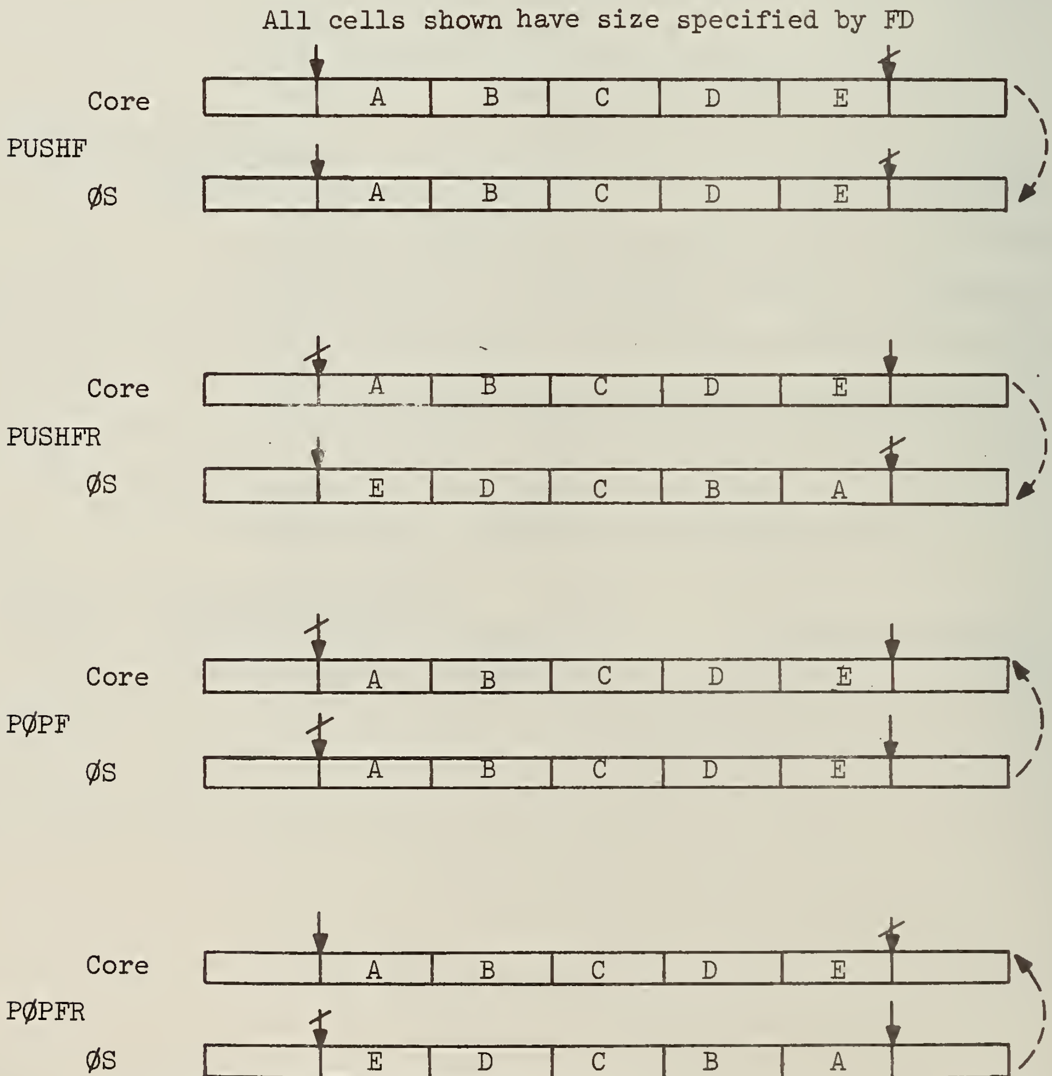


Figure 2.2.2.1.5 Pointer Manipulation Conventions

2.2.2.2 Field Comparison

These two instructions allow fields to be scanned for desired bit patterns. The SCAN and SCANM instructions are simply multi-cycle forms of the TEST and TESTM instructions respectively. If the operation is terminated by a match, the Source Pointer Register will address the matched cell. If the count is exhausted, or a flag terminates the operation, the PR will address the following cell.

2.2.2.2.1 Scan [1]

SCAN | 0 0 1 0 1 0 | F D | <Count><Øperand>$_S$

The topmost FD cell in the ØS is treated as a bit pattern. Bits in the top of the ØS and logically compared with the bit pattern of the operand. The ØS is unchanged. Status indicators are set according to the comparison results. The instruction is terminated by a count, a flag, or if a match is found.

Indicators: bounds overflow, parity check, flag match, greater than, equal, less than.

2.2.2.2.2 Scan with Mask [1]

SCANM | 0 0 1 0 1 1 | F D | <Count><Øperand>$_S$

The topmost FD cell in the ØS is treated as a bit pattern. The next-to-top cell in the ØS is treated as a mask. Bits of the mask which are '1' define corresponding bits in the cell at the top of the ØS which are logically compared with the bit pattern of the source operand. The defined data bits are compared for match or no match. The ØS is unchanged. (Effectively, the mask is AND'ed with the data cells and then logically compared with the bit pattern.) Status indicators are set according to the comparison results. The instruction is terminated by a count, a flag, or if a match is found.

Indicators: bounds overflow, parity check, flag match, greater than, equal, less than.

2.2.2.3 Decimal Conversion

These two instructions convert fields of data to and from packed decimal form.

2.2.2.3.1 Pack Numeric

		1	
PACK	0 0 | 0 0 1 1 0 0		<Count><Øperand>$_S$

The source operand specifies the address of the first byte of a field which is to be packed from a single USACII-8 numeric character per byte to two characters per byte packed decimal form in the ØS. An even number of bytes must always be specified.

Indicators: bounds overflow, parity check, invalid count.

2.2.2.3.2 Unpack Numeric

		1	
UNPACK	0 0 | 0 0 1 1 0 1		<Count><Øperand>$_D$

The ØS contains a packed decimal number. The destination operand specifies the address of the last byte +1 in core of the resulting unpacked number. The standard USASCII-8 zone field is used for every character. The count [illegible]ers to the number of packed bytes. An even number of unpacked bytes always results.

Indicators: bounds overflow, parity check.

2.2.2.4 String Manipulation

These three instructions allow displacement, character-by-character translation and editing of variable length fields (strings).

2.2.2.4.1 Move

1

MOVE | 0 0 1 1 0 0 0 0 | <Count><Øperand>$_D$<Øperand>$_S$

This instruction moves data from the source address to the destination address. The order of bytes is preserved by the instruction.

Note: All operand phrases must be short.

Programming Note: To gain efficiency, the move instruction attempts to move a double word at a time. However, if the beginning and/or ending address are not on double word boundaries, the instruction proceeds a byte at a time until a double word boundary is encountered or the count is exhausted (or a flag is encountered).

2.2.2.4.2 Translate

1

TRANS | 0 0 1 1 0 0 0 1 | <Count><Øperand>$_S$<Øperand>$_S$

The first source operand specifies the first byte of a field to be translated. The second source operand specifies the first byte of a 256 byte translation table. For each byte to be translated, the byte value is considered to be an 8-bit integer. This integer is added to the second operand address and the contents of the resulting addressed byte replaces the byte to be translated. Translation proceeds from left to right and takes place one byte at a time. All bit combinations are valid.

Note: All operand phrases must be short.

2.2.2.4.3 Edit

EDIT [0,0,1,1,0,0,1,0] [1] <Count><Operand>$_D$<Operand>$_S$

The source operand specifies a source field consisting of an even number of packed decimal characters and/or the sign codes. The destination operand specifies where the edited field will be placed. The ØS contains the pattern in reverse order: The top byte of the ØS is the first pattern byte. The count specifies the number of pattern bytes to be used. If there is no count a flag on a pattern byte terminates the instruction after that byte has been processed. The pattern is in unpacked format and may contain any valid USASCII-8 character. The execution of the edit instruction consists of editing the source field under control of the pattern (which is popped out of the ØS) and placing the result in the destination field. For a more complete description, see the description of the IBM 360 Edit instructions in IBM manual A-22-6821.

Note: All operand phrases must be short.

2.2.3 Stack Utility

This class of (3) instructions provides various stack manipulation operations. Here FD refers to the 4 cell size variants only; see Section 2.1.1.

2.2.3.1 Exchange

XCH

0 0	1 0 1 0	F D

(0 above the F D field)

The top two FD cells in the OS are exchanged. First the top two cells are POP'ed out; the top cell is then PUSH'ed back into the OS, followed by the second, effectively reversing the relative positions of the cells in the OS.

Indicators: bounds overflow.

2.2.3.2 Duplicate

DUP

0 0	1 0 0 1	F D

(0 above the F D field)

The top FD cell in the OS is PUSH'ed into the OS, duplicating itself.

Indicators: bounds overflow.

2.2.3.3 Sluff

SLUFF

0 0	1 0 0 0	F D

(0 above the F D field)

The OSP is decremented by FD, effectively "sluffing" off (storing in a non-existent location) the top FD cell in the OS.

Indicators: bounds overflow.

2.2.4 Logical Operations

All of the (10) variable cell size instructions in this class obtain their operands from the Operand Stack (OS). All are specified by a mnemonic byte only. Here, FD refers to the 4 cell size variants only, see Section 2.1.1.

(The bounds overflow indications in this section usually occur as an "underflow" -- i.e., there are not sufficient cells in the OS to complete the operation.)

2.2.4.1 Unary Logical

This set of (5) instructions operates on a single operand at the top of the ØS.

2.2.4.1.1 Zero

[O]

ZERO [0 0 0 0 0 0 | F D]

All bits (including flags) of the top FD cell in the OS are set to '0'.

Indicators: none.

2.2.4.1.2 One

[O]

ONE [0 0 0 0 0 1 | F D]

All bits (including flags) of the top FD cell in the OS are set to '1'.

Indicators: none.

2.2.4.1.3 Not

[O]

NOT [0 0 1 1 0 0 | F D]

The bitwise one's complement (including flags) of the top FD cell in the OS is formed and replaced in the OS.

Indicators: none.

2.2.4.1.4 Count Ønes

CØUNT | 0 0 0 0 1 0 | F D |

The number of bits (including flags) in the top FD cell in the OS which are 1's is counted. This cell is then pop'ed and the resultant halfword count is PUSH'ed into the OS.

Indicators: bounds overflow.

2.2.4.1.5 Bit

BIT | 0 0 0 0 1 1 | F D |

The bit address (including flags) of the first '1' (counting from the left) in the top FD cell in the OS is computed. The cell is then pop'ed and the resultant halfword count is PUSH'ed into the OS.

Indicators: bounds overflow.

2.2.4.2 Binary Logical

This set of (5) instructions operates on the top and next-to-top cells of the OS. These two cells are then POP'ed out and the result is PUSH'ed into the OS. For the instructions AND, OR, XOR, and EQV, the result is checked for zero and the equal and flags match indicators are set accordingly.

2.2.4.2.1 And

AND | 0 0 0 1 0 0 | F D |

The top two FD cells in the OS are POP'ed out. The bitwise logical 'AND' of the two cells is formed and the resultant cell is PUSH'ed into the OS.

Indicators: bounds overflow, equal, flags match.

2.2.4.2.2 Or

OR [0] 0 0 0 1 0 1 | F D

The bitwise logical 'OR' of the top two FD cells in the OS is formed, as in 'AND', above.

Indicators: bounds overflow, equal, flags match.

2.2.4.2.3 Exclusive Or

XOR [0] 0 0 0 1 1 0 | F D

The bitwise logical 'EXCLUSIVE OR' between the next-to-top FD cell and the top FD cell in the OS is formed, as in 'AND', above.

Indicators: bounds overflow, equal, flags match.

2.2.4.2.4 Equivalence

EQV [0] 0 0 0 1 1 1 | F D

The bitwise logical 'EQUIVALENCE' of the top two FD cells in the OS is formed, as in 'AND', above.

Indicators: bounds overflow, equal, flags match.

2.2.4.2.5 Compare Logically

CPRL [0] 0 0 1 0 1 1 | F D

The next-to-top FD cell is logically compared with the top FD cell in the OS. The data bits of the next-to-top cell are compared for greater than, equal to, or less than the data bits of the top cell. The flag bits are compared in their respective positions for a match condition. Status indicators are set according to the comparison results. The top cell is then POP'ed out and lost.

Indicators: bounds overflow, parity check, greater, equal, less, flags match.

2.2.5 Shift

This class of (2) instructions provides the capability of shifting a byte, halfword or word at the top of the ØS. Here, FD refers to the 4 cell size variants only, see Section 2.1.1.

2.2.5.1 Left Shift

LS | 0 1 1 0 0 0 | F D | <Count>

The top FD cell (excluding double words) in the ØS is shifted left the number of bit positions given by the pointer register specified by the count operand. Bits shifted off the left end are lost. Zeros are injected on the right. Flags are shifted in steps of 8 only.

Indicators: parity check

2.2.5.2 Right Shift

RS | 0 1 1 0 0 1 | F D | <Count>

The top FD cell (excluding double words) in the ØS is shifted right as in LS above. Zeros are injected on the left. Flags are shifted in steps of 8 only.

Indicators: parity check

2.2.6 Conditional Instructions

These two instructions are used to check the status of various indicators within the TP as specified by the mask pointer register. The indicators which may be checked and their corresponding mask bits in the value field of the mask PR are given below:

PR Bit	Condition
19	Conditional Subtraction
20	Overflow
21	Greater Than
22	Equal
23	Less Than
24	Flags Match

2.2.6.1 If

IF | 0 0 1 1 0 1 0 0 | [1]   <operand>, <operand>

The conditions specified by the value field of the PR whose name is in the tag field of the first operand are tested. If any condition is true, control is transferred to the instruction located at the address specified by the second operand phrase. If no condition is true, the next instruction in sequence will be executed.

2.2.6.2 If Not

IFN [0 0 1 1 0 1 0 1] [1] <operand>, <operand>

The conditions specified by the value field of the PR whose name is in the tag field of the first operand are tested. If any condition is false, control is transferred to the instruction located at the address specified by the second operand phrase. If no condition is true, the next instruction in sequence will be executed.

2.2.7 Utility

2.2.7.1 Location 1

LOC 0 0 0 1 0 1 0 0

The 24-bit address of the instruction (preceded by a byte of zeros) is pushed into the OS as a fullword. The flags are cleared.

Indicators: bounds overflow

2.2.7.2 Specify 1

SPECIFY 0 0 0 1 0 1 0 1 <Øperand$_1$>, ..., <Øperand$_n$>

The operations specified by the operand phrase(s) are performed on the specified pointers. At least one operand phrase must follow the mnemonic byte but there is no upper limit on the number of allowable phrases. As in the imprimitive instructions, the last bit of the first byte of an operand phrase is used as a 'Last' bit to indicate a terminal operand phrase.

Indicators: bounds overflow, CS

2.2.7.3 No Operation 1

NØP 0 0 0 1 0 1 1 0

No operation is performed. No operand phrases are allowed.

2.2.8 Arithmetic Operations

This class of (8) instructions consists of the subset of Illiac III arithmetic operations which are performed in the TP's. For these instructions, the AU is not used. For details about the flag explanation, see Section 2.3.

2.2.8.1 Unary Arithmetic

The instructions use as their single operand the number at the top of the ØS. These instructions do not check for invalid decimal numbers.

2.2.8.1.1 Negate [1]

NEG | 1 0 | 0 1 0 0 | N T |

Fixed: Form 2's complement of the number at the top of ØS.

Floating: Change the sign of the fraction of the number at top of ØS.

Decimal: Change the sign of the decimal number at top of ØS.

+ = (1011), - = (1101)

Flags: Unchanged

Indicators: [+] OV (Fixed Only)

2.2.8.1.2 Absolute Value [1]

ABS | 1 0 | 0 1 0 1 | N T |

Form the absolute value of the number at the top of the ØS.

Flags: Unchanged

Indicators: [+] OV (Fixed Only)

[+] Although the decimal representation may be invalid, no ID check will be made for unary operations.

2.2.8.1.3 Minus

MNS

1 0	0 1 1 0	N T

Form the negative of the absolute value of the number at the top of the ØS.

Flags: Unchanged

Indicators: None

2.2.8.1.4 Test Algebraically

TA

1 0	0 1 1 1	N T

Compare the number at the top of the ØS with zero and set a Greater Than (GT), Less Than (LT), or Equal Zero (EQ) indicator. Compare flag bits with zeros and set Flags Match (FM).

ØS unchanged.

Flags: Unchanged

Indicators: greater, less, equal and flags match

2.2.8.1.5 Convert Short to Long Fixed Point

CVL

1 0	0 0 0 1	0 0

Convert the specified number at top of ØS into a long fixed point number.

Flags: $F'_{0-1} \leftarrow F_{0-1}$

$F'_{2-3} = 0$

Indicators: None

2.2.8.2 Binary Arithmetic

These instructions operate on the top two numbers in the ØS. These operands are then POP'ed out and the result PUSH'ed into the ØS.

2.2.8.2.1 Fixed Point Addition [1]

ADD [1,0|1,0,0,0|N,T]

Add the top two numbers in the ØS, decrement the ØSP by cell size (of NT) and place the sum into the new top of stack position.

Before ADD [A | B]

After ADD [A + B]

Flags: F(A + B) ← F(A)

Indicators: ØV

2.2.8.2.2 Fixed Point Subtraction [1]

SUB [1,0|1,0,1,0|N,T]

Subtract the top number in ØS from the next-to-top number, decrement the ØSP by cell size (of NT) and place the difference in the new top of stack position.

Before SUB [A | B]

After SUB [A - B]

Flags: F(A - B) ← F(A)

Indicators: ØV

2.2.8.2.3 Fixed Point Algebraic Comparison [1]

CPRA | 1, 0 | 1, 0, 1, 1 | N, T |

Compare algebraically the next-to-top number in ØS with the top number in the ØS. Set GT, LT, EQ Indicators. Compare flags for match or no match and set appropriate indicator.

Decrement ØSP by cell size.

If A - B > 0, set Greater Than (GT)

If A - B = 0, set Equal (EQ)

If A - B < 0, set Less Than (LT)

If flags of A match flags of B, set Flags Match (FM)

Before CPRA | A | B | △

After CPRA | A | △

Flags: Unchanged

Indicators: greater, less, equal and flags match

2.2.9 List Processing Instructions

The list processing instructions are intended to supply the basic operations for a variety of list processing systems.

The basic list processing element is the "cell". A cell is an even number of contiguous bytes consisting of one or more pointer fields followed by an optional number of data fields. The fields can be thought of as any size, but the total length of the cell must be an even number of bytes. This stipulation is consistent with the cell alignment convention that a halfword pointer will <u>never</u> overlap a halfword boundary, i.e. all the pointers must begin on an even byte.

The purpose of this cell format is to make the list processing system as flexible as possible so that a wide variety of LP system types can be utilized. As examples, bidirectional sequencing of lists, such as those encountered in SLIP, may be performed if the two sequence links point to the previous and succeeding cells. However by having both links point to succeeding cells, one can construct and scan binary trees. Stacks can be implemented by using only one pointer field which points to the next entry in the stack. Minor modifications to these structures will allow the construction of n-ary trees and circular lists.

The basic sources of storage in the Illiac III list processing instructions are the available space segments. There is always at least one PR devoted to keeping track of available space (PR#14). In addition the programmer may dedicate other PR's to keeping track of available space if this is desired. The cells in available space may vary in size from 4 bytes on up. However a different PR must be used for each different cell size.

The cells for the structure are obtained using GET to obtain the needed cells and PUT to return them to Available Space when no longer needed. The structure can be built by using INCL and INCR, if

bidirectional lists are being formed, by the slash conventions if stacks are being used, or by combinations of ASSIGN, SPECIFY, and other Illiac III instructions for other types of structures.

In scanning through the various structures, a search is carried out using a pointer register as a "bug" (in the L^6 sense). The SR and SL instructions cause the PR to follow one of two possible pointers and loads the present location of the "bug" into the top of the OS.

In general, the OS is used to contain pertinent addresses.

It should be noted that the insert and delete instructions are by far the most specialized since they specifically assume that the list structure is to be a bidirectional list (e.g. SLIP lists). One reason for the direct implementation of these instructions was that without them SLIP implementation would be fairly inefficient when compared with Illiac III machine language. With these instructions, however, in combination with the other list processing instructions, it is virtually possible to implement a machine language level version of SLIP.

2.2.9.1 Sequence Left

SL | 0 1 0 | 0 0 0 0 1 | [1]   <Operand>

The first 4 bytes of the cell pointed to by the left link of the PR specified by the tag of the operand phrase are loaded into that PR. The top halfword of the OS is loaded with the old left link.

Indicators: bounds overflow, parity check

2.2.9.2 Sequence Right

SR | 0 1 0 | 0 0 1 0 1 | [1]   <Operand>

The first 4 bytes of the cell pointed to by the right link of the PR specified by the tag of the operand phrase are loaded into that PR. The top halfword of the OS is loaded with the old right link.

Indicators: bounds overflow, parity check

2.2.9.3 Get Cell

GET | 0 1 | 0 0 1 | 0 0 1 | [1] $\langle Operand\rangle_{AS}$

Remove the top cell from the Available Space list specified by the PR named by the tag of the operand phrase and push its address (halfword) into the OS.

Indicators: bounds overflow, parity check

2.2.9.4 Put Cell

PUT | 0 1 0 0 | 1 1 0 1 | [1] $\langle Operand\rangle_{AS}$

Put the cell whose address appears in the top halfword of the OS in the Available Space list specified by the tag of the operand phrase and then sluff this address out of the stack.

Indicators: bounds overflow, parity check

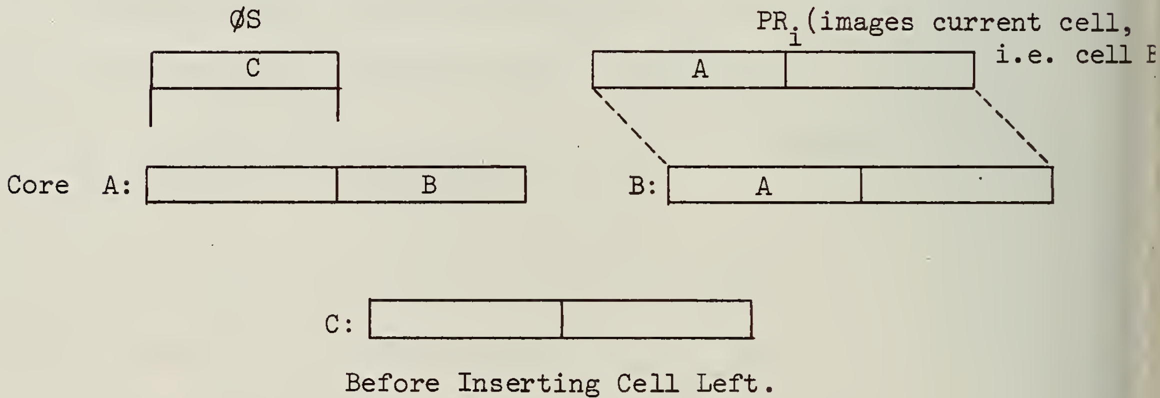
2.2.9.5 Insert Cell Left

INCL | 0 1 0 1 0 0 0 1 | [1] | <Operand>$_S$

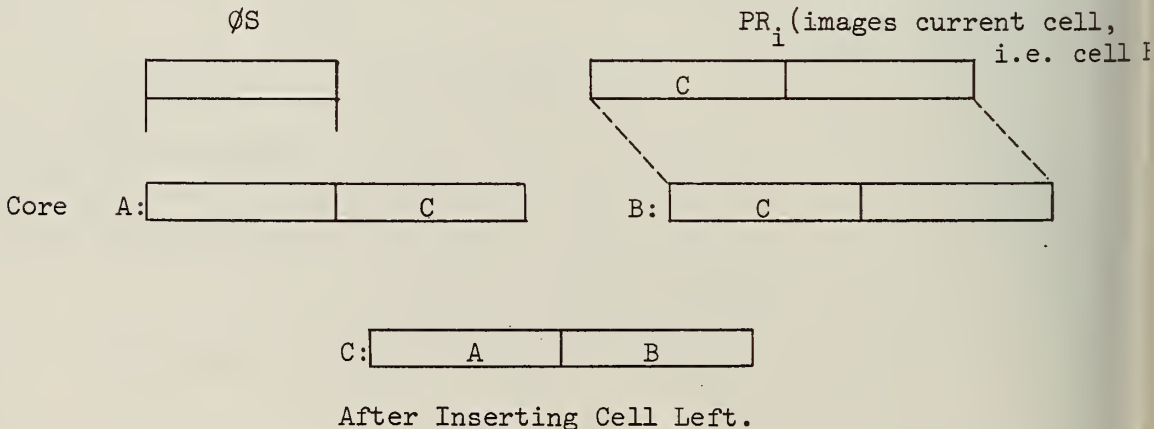
The operand phrase names the PR which contains the links of the current cell. Thus, the left (right) pointer halfword is assumed to be a backward (forward) link. The OS contains the address (halfword) of a new cell to be inserted between the current cell and the cell pointed to by the left link. The OS is then popped. (Note: the current cell (B) is not pointed to by PR_i.)

Indicators: bounds overflow, parity

The modified links are shown below:



Before Inserting Cell Left.



After Inserting Cell Left.

2.2.9.6 Insert Cell Right

INCR | 0,1,0,1,0,1,0,1 | [1]   <Operand>$_S$

The cell whose address is contained in the top halfword of the OS is inserted to the 'right' of the current cell. Link operations are symmetric with INCL. (Note: the <u>current</u> cell (B) is <u>not</u> pointed to by PR_i.)

Indicators: bounds overflow, parity

2.2.9.7 Delete Cell Left

DECL | 0,1,0,1,1,0,0,1 | [1]   <Operand>$_D$

The bidirectional list structure of INCL and INCR is assumed. The cell pointed to by the left link of the specified PR is deleted and its address is pushed into the OS. The instruction may be thought of as the inverse of INCL.

Indicators: bounds overflow, parity

2.2.9.8 Delete Cell Right

DECR | 0,1,0,1,1,1,0,1 | [1]   <Operand>$_D$

The bidirectional list structure of INCL and INCR is assumed. The cell pointed to by the right link of the specified PR is deleted and its address is pushed into the OS. The instruction may be thought of as the inverse of INCR.

Indicators: bounds overflow, parity

2.2.10 Imprimitive Instructions

The imprimitive instructions GØTØ, EXECUTE, CALL, and EXIT are used to accomplish transfers of control (temporary or permanent) and to simultaneously modify pointer stacks. Imprimitive instructions operate only on the pointer stacks (including their names): no terminal action is performed on any data. In some cases an imprimitive instruction can be considered to be an instruction whose definition must be obtained, since imprimitive instructions provide a means of calling subroutines.

An imprimitive instruction consists of an mnemonic byte optionally followed by operand phrases. Except for one bit (the LAST bit), the interpretation of an imprimitive operand phrase is identical to that of a primitive operand phrase.

2.2.10.1 Operational Description

The four imprimitive instructions can be separated into two groups: the first group consists of the GØTØ and EXIT instructions which effect permanent transfers of control; the second group consists of the EXECUTE and CALL instructions which effect temporary transfers of control. The execution of instructions which effect a temporary transfer of control includes a complete name permutation as described below in Section 2.2.10.6 to link actual parameters with formal parameters.

The following sections give detailed descriptions of each imprimitive instruction. For purposes of discussion, modification operations are REPLACEMENT, ADDITION, and CONDITIONAL SUBTRACTION. Preoperations (i.e., operations prior to effective address construction and instruction execution) are the above modifications, indirect addressing modifications, and preslash. Postoperations are the post-slash pointer stack adjustment.

A transfer of control in Illiac III can be accomplished in either of two ways: modification of the value of PR_0 (by a primitive or imprimitive instruction) or by changing the register designated PR_0 (by an imprimitive instruction). Since the location of the next instruction is always designated by the (current) value of PR_0, either of these methods will effect a transfer of control. For the EXECUTE and CALL imprimitive instructions the address of the instruction must be saved so that the postoperations may be performed when control is returned. The proper place to store this virtual address (Segment Name and Value) is in the new PR_0. Since the value of the new PR_0 will be the current instruction pointer, the return address must be stored one level below the top of the new instruction stack. Thus it is necessary that the level of the instruction pointer stack be the same when leaving a CALL'ed or EXECUTE'd instruction string as it was upon entry.

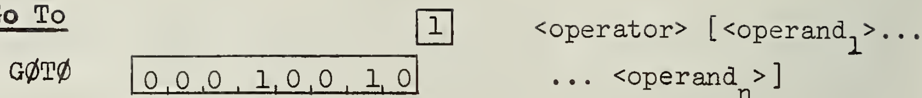
After the last operand phrase has been considered for pre-operations (i.e., at the completion of the prescan) several values must be available to complete the operations of an imprimitive instruction.

The particular values required depend on the instruction being executed, whether or not the tag of the operator phrase is zero, and whether or not there has been a conditional subtraction failure in some phrase. The values which may be needed are as follows:

1. The address of the first byte of the imprimitive instruction, denoted α_0.
2. The address of the next instruction in core, $\alpha_0 + \beta$, where β denotes the length of the imprimitive instruction.
3. The modified value of PR_0, denoted α_0' if any modification of PR_0 has occurred.

Note that any modifications made to PR_0 are made to the second level of the stack since the top level must be used to control the scan of the imprimitive instruction. After the scan is completed the extraneous value is discarded.

2.2.10.2 **Go To**



The GØTØ instruction effects a permanent transfer of control. The name of the pointer register specified by the tag of the operator phrase and the name of PR_0 are exchanged during the execution of a GØTØ instruction.

If one or more phrases specify Conditional Subtraction, then following completion of the prescan, the CS indicator will be reset to the 'OR' of the results from the individual Conditional Subtraction attempts. If no phrases specify Conditional Subtraction, the CS indicator will not be reset.

If one or more phrases failed CS, control pointer names are not swapped, the instruction is not executed, and, in addition, any modification performed on PR_0 is cancelled. The next instruction executed is the one immediately following in core.

If no phrases <u>fail</u> Conditional Subtraction (or Conditional Subtraction was not specified), normal execution of the GØTØ takes place: name swapping occurs and the transfer of control completes the instruction. The ADDITION modification of PR_0 is performed relative to the address of the first byte of the current instruction.

No postoperations are ever done for a GØTØ instruction.

Figure 2.2.10.2/1 illustrates typical register values prior to scanning the instruction. Three hardware registers are illustrated. Above, is the 4-byte Spare Buffer Register (SBR). In what follows, this register is used as two halfword registers which hold volatile information during the scan of the instruction. The value and segment name are associated with the left and right halfwords respectively of the SBR.

As shown, PR_0 and PR_i have the initial values α_0, and α_i, respectively. Here α_0 is the address of the next instruction: in this case, a GØTØ instruction.

Figure 2.2.10.2/2 illustrates the contents of important stacks following completion of the prescan. Two cases are distinguished:

a. The operator phrase specifies PR_i $(i \neq 0)$. Thus PR_0 will be renamed.

b. The operator phrase specifies PR_0. Hence PR_0 will not be renamed.

At the start of the prescan, the value and segment name of PR_0 are saved in the SBR as shown. The scan then proceeds, using PR_0 as the instruction counter. Should an operator/operand phrase specify modification of PR_0, the value of PR_0 held in the SBR will be modified. Thus, modification of PR_0 takes place with respect to the beginning of the instruction.

In Figure 2.2.10.2/2, the value of PR_0 has been incremented during the scan and is now $\alpha_0 + \beta$, where β is the instruction length. The SBR will contain the modified value of PR_0, if such a modification was specified; otherwise it will contain the original value of PR_0. Prior to exchange it must be determined whether PR_0 must be reloaded with the contents of the SBR. (PR_0 and PR_i may have different Segment Names.)

Figure 2.2.10.2/3 illustrates the register contents following execution of the instruction, provided no phrase failed Conditional Subtraction. Notice that if the operator tag was not zero, the names of PR_0 and PR_i are exchanged. No postscan, and hence, no postslashing is performed for a GØTØ instruction; the SBR will be transferred to PR_0 provided one of the three pointer modifications has successfully taken place (Section 2.1.2.7). The next instruction will then be pointed to by the new PR_0.

Figure 2.2.10.2/4 illustrates the register contents if one or more phrases failed Conditional Subtraction. In this case, pointer names have not been swapped and PR_0 contains $\alpha_0 + \beta$, the address of the next instruction in sequence.

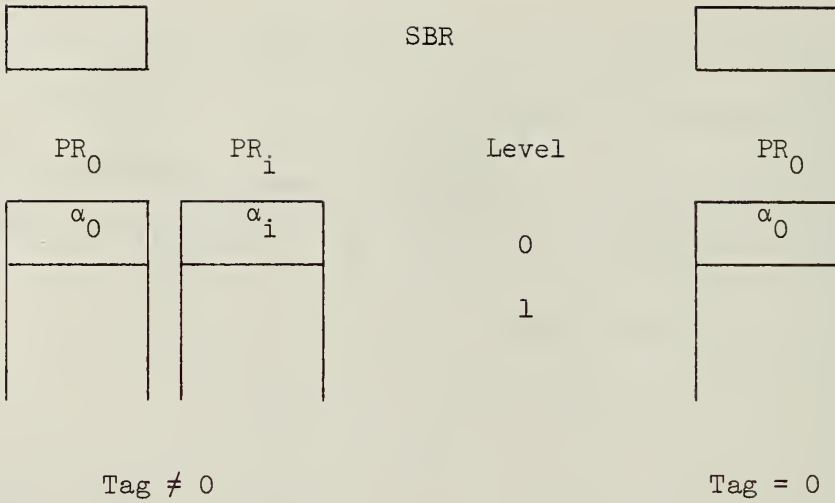


Figure 2.2.10.2/1 GØTØ: Before Prescan

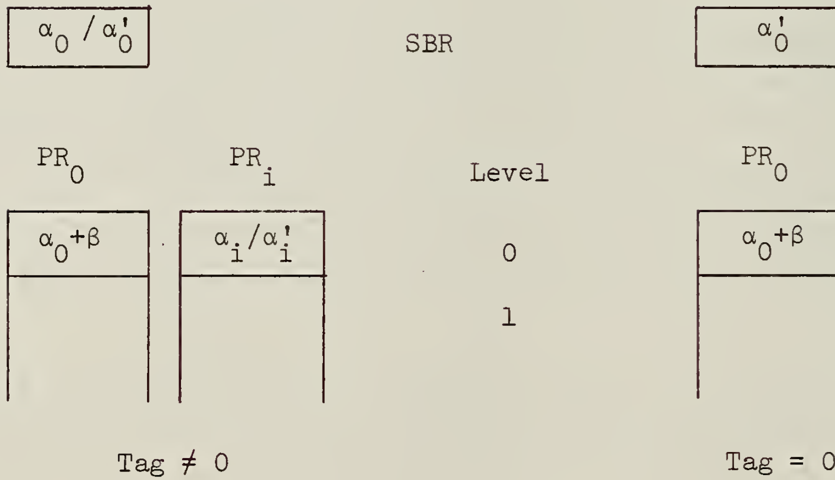


Figure 2.2.10.2/2 GØTØ: Following the Prescan

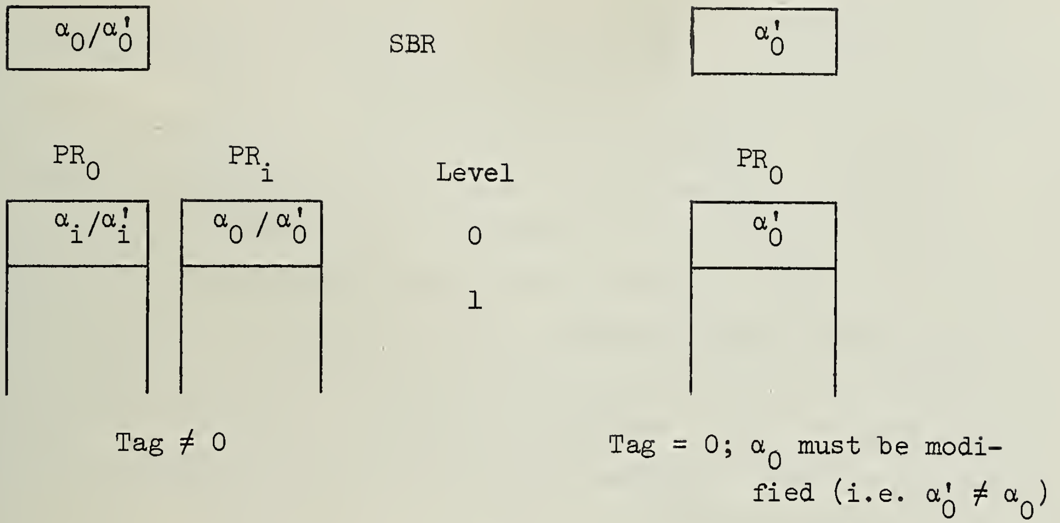


Figure 2.2.10.2/3 GØTØ: Post-execution, no CS failure

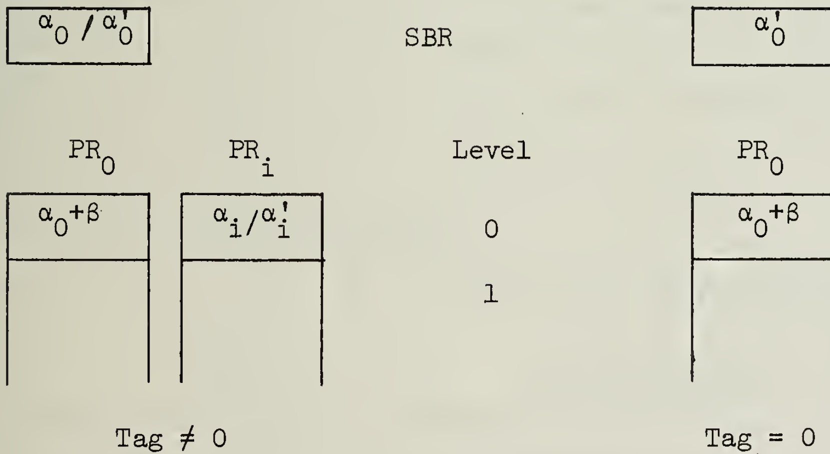


Figure 2.2.10.2/4 GØTØ: Post-execution, CS failure

2.2.10.3 Execute

EXECUTE | 0 0 0 1 0 0 0 1 | [1] | <operator>[<operand$_1$>... ...<operand$_n$>]

The EXECUTE instruction effects a temporary transfer of control. A complete name permutation occurs during the execution of an EXECUTE instruction.

If one or more phrases specify Conditional Subtraction, then following completion of the prescan, the CS indicator will be reset to the 'OR' of the results from the individual CS attempts. If no phrases specified Conditional Subtraction, the CS indicator will not be reset.

After the execution of a single (primitive or imprimitive) instruction at the address specified by the new PR_0, or immediately after the preoperations if there was a CS failure, the postoperations specified are performed on the operator and operand phrases. Then the next instruction to be executed is fetched from the location designated by PR_0. It must be remembered that ADDITION of PR_0 is always performed relative to the address of the first byte of the current instruction.

Execute Prescan:

Figures 2.2.10.3/1 through 2.2.10.3/4 pertain to the discussion following. Much of the discussion for GØTØ applies equally well to EXECUTE (and CALL).

The initial register arrangement is assumed to be the same as shown in Figure 2.2.10.2/1 and has thus been omitted.

Figure 2.2.10.3/1 shows the registers following processing of the operator phrase. The entires at level 2 represent the original values in these registers. Level 1 contains the return address; α_0 (segment name and value) have been inserted here.

In each case, PR_0 at level 0 is used to process the scan and has, in the example, been incremented by the appropriate amount. If the tag of the operator phrase is not 0, (i.e. PR_i), level 0 contains the modified value; the location to which control is ultimately to be passed. If the operator tag is 0, the modified value is stored in the SBR.

Figure 2.2.10.3/2 shows the corresponding arrangement following completion of the Prescan and Name Permutation (Section 2.2.10.6). At this point, provided no phrase failed Conditional Subtraction, the Execute bit (the leftmost bit of the new PR_0) is set ON (= '1') and control passes to α_i'(or α_0'). Thus, EXECUTES are "stackable" to any depth. Following execution of a primitive (or imprimitive) instruction, the execute bit is always examined. If it is on, or if one or more phrases failed CS, an EXIT instruction executed and control passes to the Execute postscan sequence.

Execute Post-Scan

The EXECUTE post-scan is begun immediately if any phrase failed CS or following completion of a successful EXECUTE (or CALL). It is the responsibility of this sequence to repermute the names, make any necessary stack adjustments required by the instruction, and process the post-operations of the operand phrase(s).

The initial register configuration is seen in Figure 2.2.10.3/3. Following completion of the postscan, the names are repermuted. The final register configuration is that of Figure 2.2.10.3/4.

Note: Any registers manipulated because of execution of the order itself will be restored to their original levels. If used in the meantime, the programmer must assure that the stack level is as shown before the postscan.

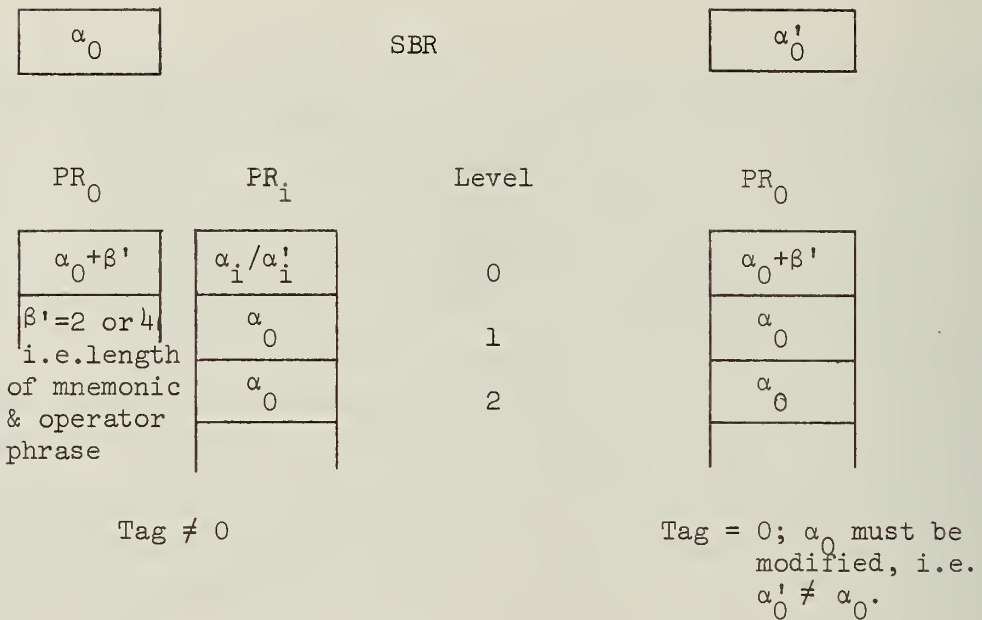


Figure 2.2.10.3/1 EXECUTE (or CALL): Following processing of the operator phrase

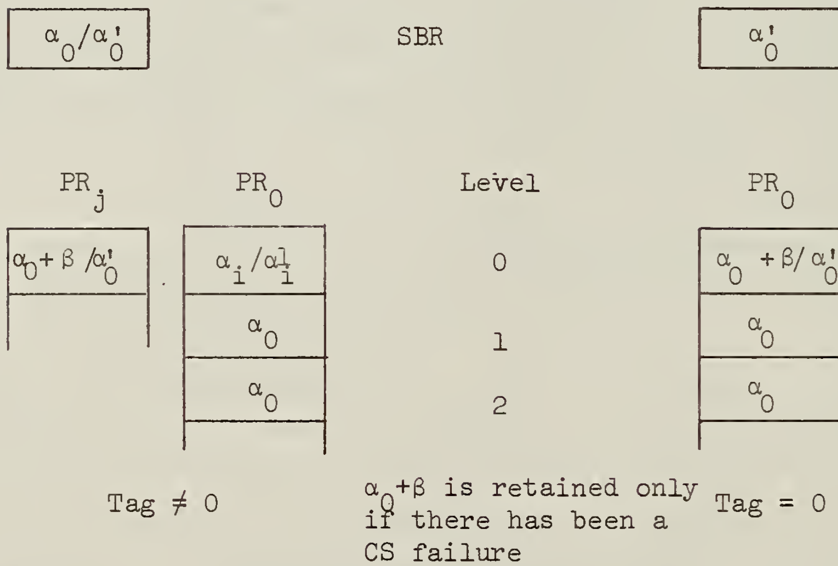


Figure 2.2.10.3/2 EXECUTE (or CALL): Following the Prescan

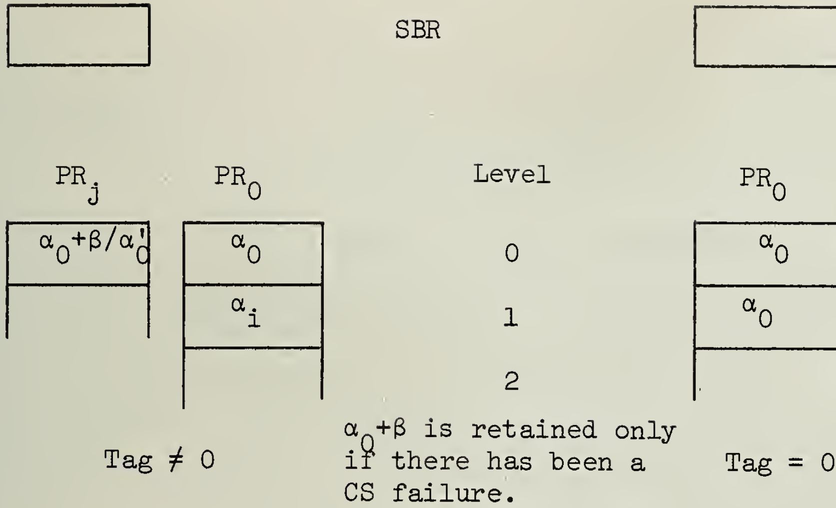


Figure 2.2.10.3/3 EXECUTE (or CALL): On Entry to the Postscan

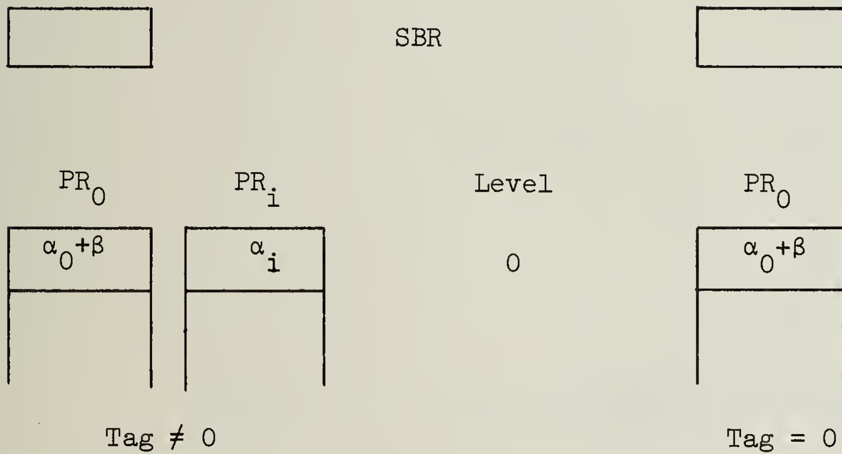


Figure 2.2.10.3/4 EXECUTE (or CALL): Following Completion of the Postscan

2.2.10.4 <u>Call</u>

CALL [0,0,0,1,0,0,0,0] [1] <operator>[<operand$_1$> ... <operand$_n$>]

The CALL instruction effects a temporary transfer of control. A complete name permutation occurs during the execution of a CALL instruction. The CALL instruction is identical to EXECUTE instruction except that for the CALL instruction, instruction execution proceeds at the location designated by the new PR_0, not for a single instruction as for EXECUTE, but until an EXIT instruction is encountered.

2.2.10.5 Exit

EXIT [0 0 0 1 0 0 1 1] [1]

The EXIT instruction is used to explicitly (implicity) return control to a CALL (EXECUTE) postscan. The instruction consists of a mnemonic byte only; no operator/operand phrases are allowed.

EXIT requires PR_0 to be arranged as is shown in Figure 2.2.10.5. Level 0 is used to perform the scan. The stack is then popped, thus returning to location α_i'. Control is passed to the EXECUTE (CALL) postscan sequence in order to complete the EXIT instruction.

PR_0	Level
α_0	0
α'_i	1

Figure 2.2.10.5 EXIT: Initial State of PR_0 Stack

2.2.10.4 Call

CALL | 0 0 0 1 0 0 0 0 | [1]   <operator>[<operand$_1$> ... <operand$_n$>]

The CALL instruction effects a temporary transfer of control. A complete name permutation occurs during the execution of a CALL instruction. The CALL instruction is identical to EXECUTE instruction except that for the CALL instruction, instruction execution proceeds at the location designated by the new PR_0, not for a single instruction as for EXECUTE, but until an EXIT instruction is encountered.

2.2.10.5 Exit

EXIT [1]

0	0	0	1	0	0	1	1

The EXIT instruction is used to explicitly (implicity) return control to a CALL (EXECUTE) postscan. The instruction consists of a mnemonic byte only; no operator/operand phrases are allowed.

EXIT requires PR_0 to be arranged as is shown in Figure 2.2.10.5. Level 0 is used to perform the scan. The stack is then popped, thus returning to location α_i'. Control is passed to the EXECUTE (CALL) postscan sequence in order to complete the EXIT instruction.

PR_0	Level
α_0	0
α_i'	1

Figure 2.2.10.5 EXIT: Initial State of PR_0 Stack

2.2.10.6 Name Permutation and Repermutation

For the CALL and EXECUTE imprimitive instructions the names of some or all of the pointer registers may be permuted. This will occur if there is no conditional subtraction failure in any phrase. The purpose of name permutation is to link actual parameters from an old control sequence to formal parameters of a new control sequence.

The renaming process is performed so that the old PR names are changed to new names beginning with 0, 1, 2 through 14. Thus the PR indicated by the tag of the operator phrase is renamed 0, the PR named by the tag of the next operand phrase is renamed 1, and so on to a maximum of 15 operand phrases.

Since the sequence of tags may not specify a complete permutation, the permutation must be completed by the TP. If the number of names in the tag sequence is i, then the names 0 through i-1 have been used as new names for the PR's. Therefore the old PR names between 0 and i-1 must be checked to make sure that all of them have been changed. If not, there will be more than one PR with the same name. Thus, each PR whose name is $\leq$ i-1 and which was not renamed during the processing of the initial tag sequence, is given a new name. These new names are taken from those used in the tag sequence which > i-1. These names are applied in numerical order to the unchanged names $\leq$ i-1.

An example will demonstrate this process more clearly. Given the tag sequence 11, 8, 3, 5, 0, the name changes will occur as follows:

"old" PR Names	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
"new" PR Names	4	5	8	2	11	3			1			0			

1) the PR now named 11 will be renamed 0
2) the PR now named 8 will be renamed 1
3) the PR now named 3 will be renamed 2
4) the PR now named 5 will be renamed 3
5) the PR now named 0 will be renamed 4

At this point the tag sequence ends so the old PR's named 0 through 4 must be checked to see if they were renamed.

6) old PR 0 has been changed ∴ do nothing
7) old PR 1 has not been changed ∴ rename it 5
8) old PR 2 has not been changed ∴ rename it 8
9) old PR 3 has been renamed ∴ do nothing
10) old PR 4 has not been renamed ∴ rename it a 11

If and when the "new" control sequence decides to return to the "old" sequence, the TP must "undo" the name permutation to get back the old PR names. This is done by rescanning the imprimitive instructions and performing the opposite operations from those given above. This process is called "repermutation".

2.2.11 System Instructions

The Illiac III Operating System has three major components: the Executive Program, the Supervisor Program and the Operating System Tables. The TP itself is most directly concerned with the Supervisor Program since it is that protion of the Operating System which has direct contact with operating tasks.

The Taxicrinic Processors have been designed so that the operations of the Supervisor Program might be facilitated. With this in mind several instructions have been implemented which can be used by the supervisor to perform its assigned functions. System instructions provided can be divided into five groups: for supervisor operation, interrupt handling, input/output, coordination, and timing respectively.

2.2.11.1 Supervisor Operation

The Supervisor instructions are directly concerned with the use and operation of the Illiac III System Supervisor. With the exception of SVC and STR, these instructions are all protected instructions and can therefore only be used when the TP is in the Supervisor mode.

2.2.11.1.1 Supervisor Call

SVC `0 0 1 1 1 1 0 0` [0]

Supervisor Call is an unprotected instruction executed by a user task to transfer control to its supervisor task. The supervisor task which will obtain control is determined by the contents of the Task Register

The current contents of all Pointer Registers is stored in the Status Segment belonging to the calling task. BR#0 is then loaded with the base descriptor of the supervisor task's Segment Table, and PR#0 is loaded with the name and virtual address of the Supervisor's SVC Validity Check procedure. Finally, the OK bits of the Associative Registers are set to zero and the Protected Instruction flip-flop is turned on.

2.2.11.1.2 Supervisor Return

SVR `0 0 1 1 1 1 0 1` [0]

The Supervisor Return is a protected instruction used by the Supervisor to transfer control to a user task. The new task which will obtain control is determined by the contents of the Task Register.

BR#0 is loaded with the base descriptor of the new task's Segment Table. The Pointer Registers are then loaded from the Status Block belonging to the new task. Finally the OK bits of the Associative Registers are set to zero and the Protected Instruction flip-flop is reset to zero.

2.2.11.1.3 Rename

RNAM | 0,0,0,0,1,1,1,0 | [1] | <Operand>$_S$ <Operand>$_{AS}$

The Rename instruction is a two operand, protected instruction used by the Supervisor to assign a given segment a known name in the Dynamic Segment List of the supervisor task's segment table. The first operand points to the given base descriptor while the second phrase indicates the available space pointer register for the Supervisor's Dynamic Segment List. (Note: this file must contain double word cells). The instruction obtains a new cell from the available space file, loads the base descriptor into this cell, and pushes the halfword address of the new cell into the ØS.

2.2.11.1.4 Sleep

SLEEP | 0,1,1,1,1,1,0,0 | [0] | <Operand>

The SLEEP instruction is a one operand, protected instruction used by the Supervisor to put the TP on which it is running into the idle state. The operand points to a halfword cell whose contents indicate the number of tenths of milliseconds the TP will remain idel before recovering and executing the next instruction in sequence. During the idel period the TP can only respond to external activate instructions given by some other TP.

2.2.11.1.5 Activate TP

ACTP | 0,0,1,1,0,1,T,P | [0]

This is a set of 4 protected instructions, one for each Taxicrinic Processor. ACTP causes the TP executing the instruction to request the Interrupt Unit to interrupt the designated TP.

2.2.11.1.6 <u>Reserve Unit</u>

RESU | 0 0 0 1 1 0 0 0 | [1] <Operand> <Operand>

The Reserve Unit Instruction, RESU, is a protected, two operand instruction used by the Supervisor to reserve a particular unit, attached to one of 18 local exchanges on the Exchange Net, to one of 6 processors. The first operand indicates a Pointer Register containing a number from 0 to 5 and indicates the processor port involved in the reservation. The second operand indicates a Pointer Register containing a number from 1 to 18 and indicates the local exchange involved in the reservation.

This instruction is necessary for those cases in which a particular processor desires to use a particular unit over a long period of time without interference from other processors. This may be necessary because of minimum response time requirements (e.g. in the case of an AU) or because intermediate data pertinent only to a particular task will be contained in the unit over a long period of time (e.g. in the case of a PAU).

2.2.11.1.7 Load Task Register

LTR [0,0,1,1,1,0,1,0] [0]

This protected instruction pops the top halfword of the Operand Stack and loads this name or the new Task Name into the TP's Task Register.

2.2.11.1.8 Store Task Register

STR [0,0,1,1,1,0,1,1] [0]

This instruction pushes the Task Name contained in the TP's Task Register in the top halfword of the Operand Stack.

2.2.11.2 Interrupt Handling

Taxicrinic Processors must be able to respond to two types of interrupt conditions: local interrupts and distal interrupts. The local interrupts concern conditions originating within the TP or as a direct consequence of a TP command to a unit: AU, PAU or Core. These interrupts are also called traps. Examples include loss of significance in a floating point AU operation, an illegal plane address in a PAU instruction, or a bounds overflow in addressing core.

Distal interrupts are caused by some external processor and are routed to a TP through the interrupt unit. Examples of this type of inter-processor communication include Active TP instructions, I/O interrupts, etc.

Interrupts are processed in several stages. Upon detection of an interrupt, it is the responsibility of the Taxicrinic Processor to save the interrupt information and the current hardware status of the machine in the Interrupt Storage Segment, and to transfer control to the current task's Interrupt Handler Procedure. This first step is entirely hardware implemented.

The Interrupt Handler Procedure, in turn, determines the type of interrupt which has occurred on the basis of the information stored in the Interrupt Storage Segment and transfers control to the appropriate procedure which will process the interrupt. NOTE: This latter processing procedure may be either a routine belonging to the present task or it may be a Supervisor routine. Interrupts do not necessarily imply Supervisor intervention.

The following two unprotected instructions allow tasks to utilize the interrupt system. An extensive discussion of interrupt handling will be found in Section 4.

2.2.11.2.1 Set Interrupt Mask

SIM [0,1,1,1,1,1,1,0] [0] Imm <operand>

This instruction loads the Interrupt Mask Register from the value field of the pointer register indicated by the operand field. A one set into a given position of the mask inhibits immediate recognition of the corresponding interrupt.

To minimize cascading of interrupts, only the Immediate Option is allowed. Execution of the instruction also momentarily suppresses distal interrupts.

2.2.11.2.2 Interrupt Return

INRT [0,0,1,1,1,0,0,0] [0]

The Interrupt Return instruction, INRT, has no operands but uses PR#1 as an implied operand, pointing to the Interrupt Storage Block containing the return status information. The execution of this instruction will cause the TP hardware to restore its status to the state contained in this storage block and also to return the storage block itself to the Interrupt Storage Segment's available space list.

2.2.11.3 Input/Output

I/O instructions enable the Supervisor to control I/O processing: typically an interrupt command[1] is constructed and sent by a TP under supervisory control to the IOP. This interrupt command is then interpreted and executed in the IOP.

The following paragraphs describe the basic procedure for executing an I/O process. Illustrated is the usage of some interrupt commands with major emphasis given to the point-in-time relationships of TP and IOP channel functions.

A Start I/O interrupt is used to initiate an I/O process. To perform this operation, it is necessary for the programmer or Supervisor to:

1) Establish an I/O program in the command area.
2) Establish a list of descriptors in that part of the descriptor area assigned to the channel.
3) Load the channel name(s) and the address of the first command of the program into the Start I/O command word.
4) Issue the Start I/O Interrupt Command to IU, which in turn will direct it to the correct IOP.
5) Test the Interrupt Acknowledgment, IA, from the IOP for success or failure in initiating the new I/O process.

An '0' in the first flag of the Interrupt Acknowledgment indicates that the new I/O process has been successfully initiated. A '1' indicates failure, and the remainder of the IA should be examined to determine why the desired operation was not initiated.

Between the time a Start I/O is issued by the TP, and the time the TP is released by the return of the Interrupt Acknowledgment, the IOP performs many functions:

---

1 In this manual, an instruction is executed in a TP, a command is executed in an IOP, and an order is executed by a peripheral device or device controller.

1) It removes the Interrupt Command from the Exchange Net.
2) It decodes the command.
3) In the case of Start I/O, it checks to see if the requested channel(s) are able to start a new process. If so, it puts a SUCCESS flag in the IA, and sends it to the requesting TP via the Exchange Net and the Interrupt Unit.

If a Start I/O results in a SUCCESS condition, the channel registers necessary for program execution are loaded from the command area and the descriptor area. A successful program start does not necessarily imply that the device will begin successfully. Actual channel operation begins when the channel attempts to select the device over the I/O interface by propagating the device address. This operation and possible further data-transfer operations continue independently of the TP until either an END-flag descriptor occurs in the I/O program, the TP issues a HALT interrupt, or an error occurs. After the termination of an I/O process, the IOP requests an End Status Interrupt of any available TP. This interrupt alerts the original program of the end of its I/O process. The status of individual I/O operations can be surveyed by checking the descriptor area. Then, if conditions are propitious, new I/O process or processes can be started on the freed channels.

An extensive discussion of input/output will be found in Volume III of this manual.

2.2.11.3.1 Start I/O

SIO [0,1,0,0,0,0,1,0] [1] <operand>

This protected instruction causes the IOP to initiate (or attempt to initiate) an I/O data transfer. The operand phrase identifies the full word "interrupt command" specifying the Start I/O operation. For interpretation of this latter, see Section 3.1.3.1. The associated Interrupt Acknowledgment and End Status Interrupt formats are discussed in Section 3.1.1.

2.2.11.3.2 Halt I/O

HIO [0,1,0,0,0,0,1,1] [1] <operand>

This protected instruction causes the IOP to terminate execution of an I/O program. The operand phrase identifies the full word "interrupt command" specifying the Halt I/O operation. For interpretation of the latter, see Section 3.1.3.2.The associated Interrupt Acknowledgment and End Status Interrupt formats are discussed in Section 3.1.1.

2.2.11.3.3 Load IOP Base Register

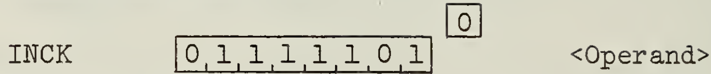
LIBR [0,1,0,0,0,0,0,0] [1] <operand>

This protected instruction loads the Segment Table Base Register (STBR) of the IOP. All segments employed by the IOP in the execution of its 8 independent programs, including the Command Segment and the Descriptor Segment, are identified by Segment Table entries: hence the central importance of the STBR.

The operand phrase identifies the full word "interrupt command" specifying the LIBR operation. The associated Interrupt Acknowledgment is discussed in Section 3.1.1.

2.2.11.4 Coordination

2.2.11.4.1 Increment and Check



The Increment and Check instruction is used to update control double words of the specified format given below:

Initial Value Field	Increment Value Field	Maximum Value Field	Pointer Value Field

Figure 2.2.11.4 Control Double Word Format

The instruction accesses the control double word at the operand address and then, without releasing the Exchange Net, writes in a modification of the control double word. This ensures that no other processor can access the control double word until after the modification has been completed. The unmodified value of the pointer value field is pushed into the OS.

The modification made is based on the contents of the various fields in the control doubleword:

1) The pointer value field is incremented by the value contained in the increment value field. If this new value is not equal to the maximum value field, the incremented value is stored in the pointer value field.
2) If the incremented value <u>is</u> equal to the maximum value field, the pointer value field is set equal to the initial value field.

3) In either case, the EQ is set according to whether the incremented pointer value and maximum value fields are equal or not (GT and LT remain unchanged).

This instruction has several uses:

- to implement loops if conditional subtraction is not desired;
- to provide control for access to "critical sections";
- as a "status indicator" for circular buffers. This is the procedure used to control the interrupt storage during interrupt;
- to provide "semaphore" instructions for parallel processing.

2.2.11.4.2 Link

LINK [0,0,0,0,1,1,1,1] [1] <Operand> <Operand>

The LINK instruction is a two operand instruction designed to perform dynamic segment linking. The first operand points to a linkage table while the second operand designates a PR which contains the relative address of the desired external reference within the linkage table and into which the proper linked address will be loaded. If the linkage table entry indicated by the first operand is not loaded with a valid linked address, a linkage interrupt will be performed.

2.2.11.4.3 Who

WHO [0,0,1,1,1,1,1,1] [0]

The WHO instruction pushes the physical name (Note: not task name) of the TP executing the instruction as a halfword integer into the top of the Operand Stack.

2.2.11.5 Timing

Illiac III maintains two types of system clocks. The accounting clock is a continuously running 24-hour clock which is slaved to the 60 cycle power. The interval timers (one for each processor) are resetable timers which are started when set and cause an interrupt of the associated processor when the time expires.

The accounting clock is maintained by the interrupt unit and may be sampled by a Read Clock (RDCLK) instruction. When a read clock instruction is executed, the interrupt unit is requested as if an interrupt were to be processed. The interrupt processor will instead return a word, called a system clock word, to the requesting terminal.

The accounting clock is a 24-bit binary counter incremented once every 1/120th of a second and cannot be reset (except manually after maintenance periods). The clock will automatically recycle every 24 hours.

On-the-other-hand the interval timer assigned to a processor is started whenever it is set to a non-zero value. The clock is a 16 bit binary counter decremented once every microsecond. When the count reaches zero, a local interrupt is initiated in the governed processor.

2.2.11.5.1 Read Clock

RDCLK | 0 1 1 0 0 0 0 0 | [1]

This instruction is used to read the 24-hour accounting clock in the Interrupt Unit. Any TP may read the clock. The time is right-adjusted in a fullword and pushed into the ØS. (Also see Sec. 4.2.1 Read Clock (RDCLK) of IU Manual, Report 386.)

2.2.11.5.2 Set Timer

STTIM | 0 1 1 0 0 0 1 0 | [1]

This instruction is used to set the timer (internal to the TP executing the instruction). The value is popped (as a halfword) from the ØS of the TP executing the instruction and loaded into the interval timer.

2.2.11.5.3 Read Timer

RDTIM | 0 1 1 0 0 0 1 1 | [1]

This instruction is used to read the interval timer. The time as a halfword is pushed into the ØS of the TP executing the instruction.

2.3 Instructions Executed by Arithmetic Units

2.3.1 Arithmetic Data Formats

There are four types of arithmetic data used in Illiac III. The attributes of these data are as follows:

Base = binary or decimal

Scale = fixed point or floating point

Mode = real

Precision = 16 bit integer, or
32 bit signed integer, or
56 bit signed fraction
and 7 bit characteristic, or
14 decimal digit signed integer.

These attributes have been combined to form four different number types as described in the remainder of this section.

Although not illustrated in the previous figures, a flag bit is associated with each byte of the four number types described. Operands used in arithmetic operations may have flags set, and thus the question arises as to the flag setting of an arithmetic result. The flag bits of numbers produced by arithmetic operations will have the following significance.

a) For unary operations (other than number type conversions) the flags of the operand are unchanged.

b) For operations with two or more operands and no error conditions, the flag setting of the result is the flag setting of one of the operands.

c) For comparison instructions, the flags transmit the result of the comparison from the Arithmetic Unit (AU) to the Taxicrinic Processor (TP) via the Exchange Net (XN).

d) For operations resulting in error conditions, the flags transmit the type error condition from the AU to the TP via the XN.

The significance of arithmetic flags is described further in Sections 2.3.2 and 2.3.3.1.

2.3.1.1 <u>Short Fixed Point</u>

Base = binary

Scale = fixed

Mode = real

Precision = 16 bit, unsigned integer if an address.

15 bit, signed integer otherwise.

A short fixed point number is a half word binary integer which is treated as signed or unsigned depending upon its use. When used as an address, a short fixed point number will always be considered positive and since all 16 bits may be required to represent the magnitude, no sign bit is explicitly specified.

Addition of addresses, as performed in the pointer modification operation ADDITION, is computed Mod (2^{16}), in effect allows implicit address subtraction. Subtraction of addresses, as performed in the pointer modification operation CONDITIONAL SUBTRACTION, is performed as a two's complement subtraction, Mod 2^{16}. The results will always be treated as a positive integer.

Numbers of the short fixed point type may also be used for other then addresses in any of the 13 arithmetic operations. In this case, the most significant (MS) bit will be treated as a sign bit and a negative number will be represented in two's complement form with a sign bit of 1. The range of these integers is -2^{15} to $+(2^{15} -1)$ i.e., -32,768 to +32,767 and overflow will be checked.

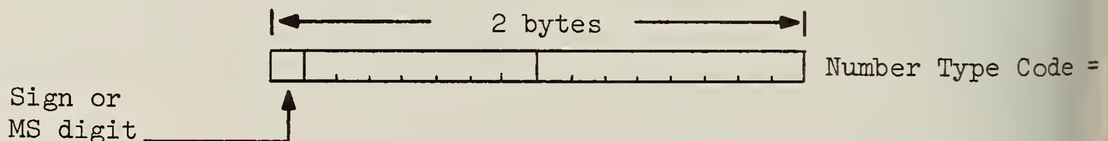


Figure 2.3.1.1 Short Fixed Point Format

2.3.1.2 Long Fixed Point

Base = binary

Scale = fixed

Mode = real

Precision = 31 bit, signed integer

A long fixed point number is a full-word signed integer. Positive numbers are represented in true binary notation with a sign bit of zero. Negative numbers are represented in two's complement notation. The range of these integers is -2^{31} to $+(2^{31}-1)$, i.e., -2,147,483,648 to +2,147,483,647 and overflow will be checked.

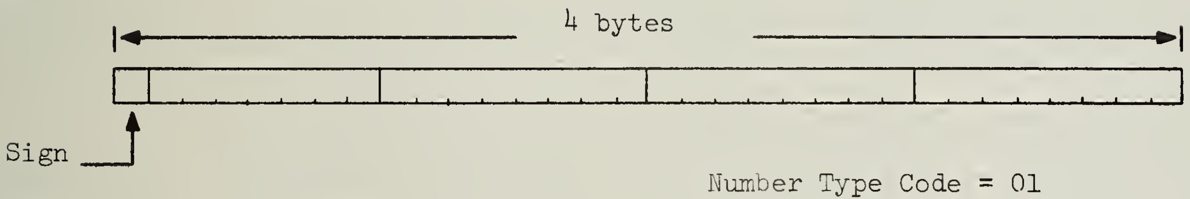


Figure 2.3.1.2 Long Fixed Point Format

2.3.1.3 Floating Point

Base = binary

Scale = floating

Mode = real

Precision = 56 bit, signed fraction
7 bit characteristic

Floating point numbers are a double word in length, subdivided as indicated below:

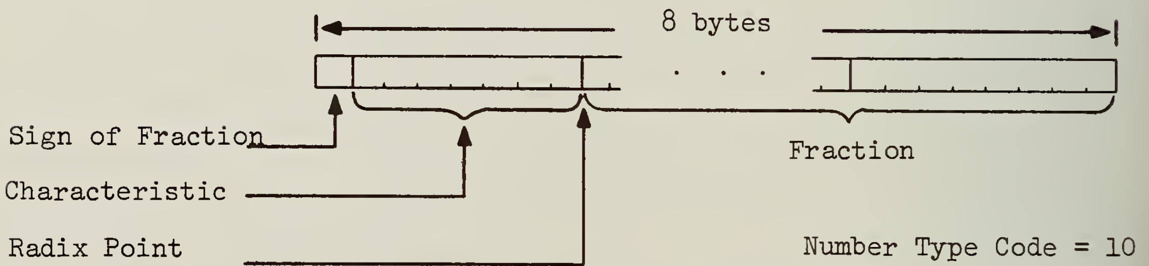


Figure 2.3.1.3 Floating Point Format

The first bit is the sign of the fraction. The fraction is always in true representation, i.e., the fraction of negative numbers is carried in positive form. The fraction is expressed in base 16, hexadecimal form and therefore consists of 14 hexadecimal digits. In hexadecimal representation, the characteristic represents the power to which 16 must be raised to express the true magnitude of the number. The 7 bits of the characteristic are treated as an excess 64 number with range -64 to +63 corresponding to the binary values 0 through 127, respectively. The characteristic zero, for example, is represented as 1000000.

A floating point number in main store or produced as the result of a floating point arithmetic operation will always be normalized. For the hexadecimal representation, this means that the fraction will have a non-zero, high-order hexadecimal digit. This type normalization permits the three high order bits of a normalized number to be zero. Normalization is not programmable.

Under the normalization described above, the range covered by this notation is 16^{-65} to $(1 - 16^{-14}) \times 16^{63}$, which is approximately 5.4×10^{-79} to 7.2×10^{75}. The binary representation of these maximum and minimum values are shown in Table 1.2.3.

A floating point zero will be represented as a number with the most negative, zero fraction, and positive sign as indicated in the figure below. All zero results will be returned from the AU in this form.

DESCRIPTION	NUMBER	POWER OF 16	S	CHAR	FRACTION
Maximum positive number	$+7.2 \times 10^{75}$	$= (1-16^{-14}) \times 16^{63}$	0	1111111	All 1's
1.0	1.0×10^{0}	$= 1/16 \times 16^{1}$	0	1000001	00010...0
0.5	0.5×10^{0}	$= 1/2 \times 16^{0}$	0	1000000	10000...0
Minimum positive number	5.4×10^{-79}	$= 16^{-1} \times 16^{-64}$	0	0000000	000100...0
True zero	+0.0	$= 0 \times 16^{-64}$	0	0000000	All 0's

Table 2.3.1.3 Examples of Floating Point Representation

2.3.1.4 Decimal

Base = decimal

Scale = fixed

Mode = real

Precision = 14 digit, signed integer

The alphanumeric representation of decimal digits uses the USASCII code extended to eight bits. The high order 4 bits are designated the zone and are 0101 for decimal digits. The low order 4 bits are the binary encoding 0000 - 1001 of the digits 0 - 9 respectively.

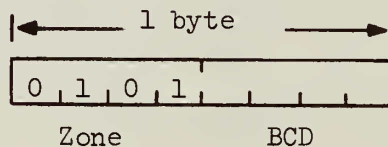


Figure 2.3.1.4/1 USASCII Digit Format

A plus sign (+) is represented as 1011 and a minus sign (-) as 1101 in the right half of the left-most byte of the number, i.e. in the half byte designated "S" in Figure 2.3.1.4/2.*

---

*Definition of USASCII-8 taken from IBM System/360 Principles of Operation, File No. S360-01, Form A22-6821-7, p. 150.1.

Decimal operands, i.e., decimal numbers to be sent to an arithmetic unit, must be in a packed or unzoned format with two digits rather than one digit per byte. A decimal number consists of a double word subdivided into BCD digits and a 4 bit sign as shown below:

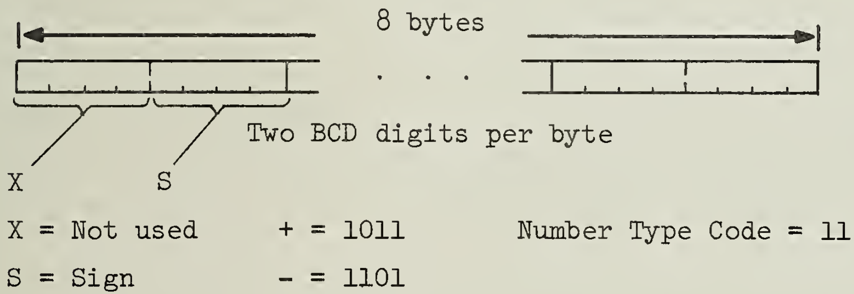


Figure 2.3.1.4/2 Decimal Number Format

Alphanumeric, or zoned format, may be converted to the above format by use of the PACK instruction of the TP.

The decimal number zero may be either plus or minus, but in either case an Equal Zero (EQ) indicator will be turned on when it is tested with a Test Algebraic (TA) instruction.

2.3.2 Arithmetic Instructions

This section describes the arithmetic instructions. The arithmetic instructions which are executed in a Taxicrinic Processor are included in Section 2.2.8. An asterisk(*) by a name denotes an order which will not be available in the initial version of the Illiac III Arithmetic Units. Provisions are being made however for their eventual incorporation, either by additional hardware or as programmed macro-instructions. The following conventions are used in this section.

a. "Fixed" includes both short and long fixed point format unless otherwise noted.

b. △ indicates the location of the Operand Stack Pointer (ØSP).

c. The abbreviations used for Indicators are as follows:

   ØV = Overflow
   LS = Loss of Significance
   GT = Greater Than
   EQ = Equal
   LT = Less Than
   FM = Flag Match
   UN = Underflow
   ID = Invalid Decimal Data

   The exceptional condition indicators are described in detail in Section 2.3.3.1.

d. The flags of results of unary operations other than number type conversions are unchanged.

e. The "flag" convention for binary and multi-cycle arithmetic is of the form:

   F(W) ← F(X)

   where F = "the flags of"
   X = the operand next to the top operand
   W = the result produced
   ← = "replaced by"

f. "Flag" convention for the conversion instructions is of the form:

$$F'_{i-j} \leftarrow F_{m-n},$$

where the flags within a number type are numbered consecutively 0, 1, ..., 7 from left to right.

F_{m-n} = the mth through nth flag of the original, unconverted number.

F'_{i-j} = the ith through jth flag of the resultant converted number.

$\leftarrow$ "replaced by"

g. Note that when an arithmetic computational error occurs, the flag setting of the result is not a copy of the flags of the operand, but is rather an indication of the type error which has occurred. (See Section 2.3.3.1).

Arithmetic Indicators include two types of indicators: comparison indicators and computational condition indicators.

Since all except fixed point comparisons (CPRA) are executed in an arithmetic unit, provisions have been included to return the results of the comparison (GT, LT, EQ, FM) to the Taxicrinic Processor. This transfer is accomplished by returning a floating point zero with the flags set to indicate the result of the comparison.

When this pseudo-result, zero, is received by the Taxicrinic Processor, the flags indicating the results of the comparison set indicator flipflops which may be tested by a subsequent instruction. For fixed point comparisons and test algebraic (TA), both of which are executed solely in a taxicrinic processor, the indicator flipflops are set directly.

The comparison indicators are designated as follows:

EQ = Equal Zero

GT = Greater Than

LT = Less Than

FM = Flags Match

The flags of a result which set these indicators are called comparison indicator flags, and are described in Table 2.3.2/1.

The comparison indicator flags are assigned as follows:

	Flag of Byte Number
GT	5
EQ	6
LT	7
FM	8

Byte numbering is 0 to 7, left to right.

Table 2.3.2/2 is a summary of the arithmetic unit order code.

Table 2.3.2/1 Comparison Indicators

Indicator And Description	Orders In Which Indicator May Occur	*Pseudo Result Returned From AU
GT - Greater Than		
A > 0	TA	None
A > B	CPRA	GT = 1 ZERØ
EQ - Equal		
A = 0	TA	None
A = B	CPRA	EQ = 1 ZERØ
LT - Less Than		
A < 0	TA	None
A < B	CPRA	LT = 1 ZERØ
FM - Flags Match		
Flags of A = 0, FM = 1	TA	None
Flags of A ≠ 0, FM = 0		
Flags of A = Flags of B, FM = 1	CPRA	FM = 1 ZERØ
Flags of A ≠ Flags of B, FM = 0		

*An arithmetic unit is used only for Decimal and Floating CPRA. TA is executed in the taxicrinic processor for all number types. In all cases the operands in the ØS are not changed.

Table 2.3.2/2 ILLIAC III Arithmetic Unit Order Code

ORDER		NUMBER TYPE			
Mnemonic	Instruction Variant	Short Fixed	Long Fixed	Floating	Decimal
(None)	0000	*	*	*	*
CVL	0001	TP	*	10	11
CVF	0010	00	01	*	11
CVD	0011	00	01	10	*
NEG	0100	TP	TP	TP	TP
ABS	0101	TP	TP	TP	TP
MNS	0110	TP	TP	TP	TP
TA	0111	TP	TP	TP	TP
ADD	1000	TP	TP	10	*
(None)	1001	*	*	*	*
SUB	1010	TP	TP	10	*
CPRA	1011	TP	TP	10	*
MPY	1100	00	01	10	*
POLY	1101	*	*	10	*
DIV	1110	00	01	10	*
(None)#	1111	*	*	*	*

*Not defined. No operation will take place.

TP - This operation performed in Taxicrinic Processor.

- IV = 1111 is used to indicate the final coefficient in a POLY order.

2.3.2.1 Add [1]

ADD [1 0 | 1 0 0 0 | N T]

Add the top two numbers in the ØS, decrement the ØSP by cell size (of NT) and load the sum into the new top of stack position.

Before ADD [A | B] △

After ADD [A + B] △

Flags: F(A + B) ← F(A)

Fixed: Indicators: ØV (Executed in TP)

Floating: Indicators: ØV, UN, LS

*Decimal: Indicators: ØV, ID

2.3.2.2 Subtract [1]

SUB [1 0 | 1 0 1 0 | N T]

Subtract the top number in ØS from the next-to-top number, decrement the ØSP by cell size (of NT) and place the difference in the new top of stack position.

Before SUB [A | B] △

After SUB [A - B] △

Flags: F(A - B) ← F(A)

Fixed: Indicators: ØV (Executed in TP)

Floating: Indicators: ØV, UN, LS

*Decimal: Indicators: ØV, ID

2.3.2.3 Multiply

MPY [1,0|1,1,0,0|N,T]

Multiply the top number (multiplicand) in the ØS by the next-to-top number (multiplier).

Before MPY [A | B]△

Fixed: The most significant part of the product $(A \cdot B)_M$ replaces the multiplier; the least significant part $(A \cdot B)_L$ replaces the multiplicand. Cell size of result is twice CS of operands.

After MPY [$(A \cdot B)_M$ | $(A \cdot B)_L$]△

Flags: $F[(A \cdot B)_M] \leftarrow F[A]$, $F[A \cdot B)_L] = 0$

Indicators: ØV

Floating: The normalized result replaces the multiplicand, and the ØSP is decremented by CS.

After MPY [$A \cdot B$]△

Flags: $F[A \cdot B] \leftarrow F[A]$

Indicators: ØV, UN

*Decimal: The unnormalized result replaces the multiplicand, and the ØSP is decremented by CS. As both operands and the result are double words, the sum of the number of significant digits in the operands must be ≤ 14 to prevent overflow.

After MPY [$A \cdot B$]△

Flags: $F[A \cdot B] \leftarrow F[A]$

Indicators: ØV, ID

*A double length result is returned for fixed multiply so as to permit fixed point numbers to be interpreted as either a fraction or integer. Overflow of the single length boundary is checked and a flag is set if it occurs; however, a Bogus Result interrupt is not generated.

2.3.2.4 Divide

DIV `| 1 0 | 1 1 1 0 | N T |`

Divide the next-to-top number (dividend) by the top number (divisor) in the ØS.

Before DIV `| A | B |` △

Fixed: The quotient replaces the dividend and the remainder replaces the divisor.

After DIV `| A/B | Remain |` △

Flags: F(A/B) ← F(A), F(Remainder) = 0

Indicators: ØV

Floating: The quotient replaces the divident, and the ØSP is decremented by CS.

After DIV `| A/B |` △

Flags: F(A/B) ← F(A), F(Remainder) = 0

Indicators: ØV, UN

Decimal: The quotient replaces the dividend and the remainder replaces the divisor.

After DIV `| A/B | Remain |` △

Flags: F(A/B) ← F(A), F(Remainder) = 0

Indicators: ØV, ID

Remainder has same sign as dividend (A-Op) except zero is always positive.

2.3.2.5 Compare Algebraically

1

CPRA

1 0	1 0 1 1	N T

Compare algebraically the next-to-top number in ØS with the top number in the ØS. Set GT, LT, EQ Indicators. Compare flags for match or no match and set appropriate indicator.

Decrement ØSP by CS.

If A - B > 0, set GT

If A - B = 0, set EQ

If A - B < 0, set LT

If flags of A match flags of B, set FM

Before CPRA

A	B

After CPRA

A

Flags: Unchanged

Fixed: Indicators: GT, LT, EQ, FM (Executed in TP)

Floating: Indicators: GT, LT, EQ, FM,

*Decimal: Indicators: GT, LT, EQ, FM, ID.

2.3.2.6 Convert to Decimal

[1]

CVD | 1,0 | 0,0,1,1 | N,T |

Convert the specified number on top of ØS into a packed (2 BCD/byte) double word decimal number.

Short Fixed: Flags: $F'_{0-1} \leftarrow F_{0-1}$

$F'_{2-7} = 0$

Indicators: None

Long Fixed: Flags: $F'_{0-3} \leftarrow F_{0-3}$

$F'_{4-7} = 0$

Indicators: None

Floating: Flags: $F'_{0-7} \leftarrow F_{0-7}$

Indicators: OV

Decimal: NØP

2.3.2.7 Convert to Floating Point

[1]

CVF | 1,0 | 0,0,1,0 | N,T |

Convert the specified number at top of ØS into a normalized floating point number.

Short Fixed: Flags: $F'_{0-1} \leftarrow F_{0-1}$

$F'_{2-7} = 0$

Indicators: None

Long Fixed: Flags: $F'_{0-3} \leftarrow F_{0-3}$

$F'_{4-7} = 0$

Indicators: None

Floating: NØP

Decimal: Flags: $F'_{0-7} \leftarrow F_{0-7}$

Indicators: ID

2.3.2.8 Convert to Long Fixed Point

CVL | 1,0 | 0,0,0,1 | N,T | [1]

Convert the specified number at top of ØS into a long fixed point number.

Short Fixed: Flags: $F'_{0-1} \leftarrow F_{0-1}$ (Executed in the TP)

$F'_{2-3} = 0$

Indicators: None

Long Fixed: NØP

Floating: Flags: $F'_{0-3} \leftarrow F_{0-3}$

F_{4-7} are lost

Indicators: ØV

Decimal: Same as above

Indicators: ØV, ID

2.3.2.9 Polynomial Evaluation

POLY | 1,0 | 1,1,0,1 | N,T | [1]

The polynomial of the form

$a_n x^n + a_{n-1} x^{n-1} + \ldots a_1 x + a_0$ is evaluated.

To specify the polynomial, the coefficients a_0 through a_n are pushed into the ØS in that order. The value of x is then pushed in followed by the degree n, a short fixed point number.

Before POLY | a_0 | ≈

≈ | a_{n-1} | a_n | x | n | △

After POLY | Ans. | △

Flags: F(Result) ← F(X)

*Fixed: Indicators: ØV,

Floating: Indicators: ØV, UN, LS

*Decimal: Indicators: ØV, LS, ID

2.3.3 Exceptional Conditions for Arithmetic Instructions

2.3.3.1 General

When a computational condition such as an overflow occurs in an arithmetic unit, this information must be returned to the taxicrinic processor. As with comparison operations (Section 2.3.2), the flags of a result are used to return this condition information, and to set indicator flipflops which may be tested with subsequent instructions. The bogus result produced is returned to the Taxicrinic Processor, not as a copy of the flags of an operand, but rather as an indication of the condition which has occurred. The fact that an error has occurred is included with the Exchange Net transmission in the control byte.

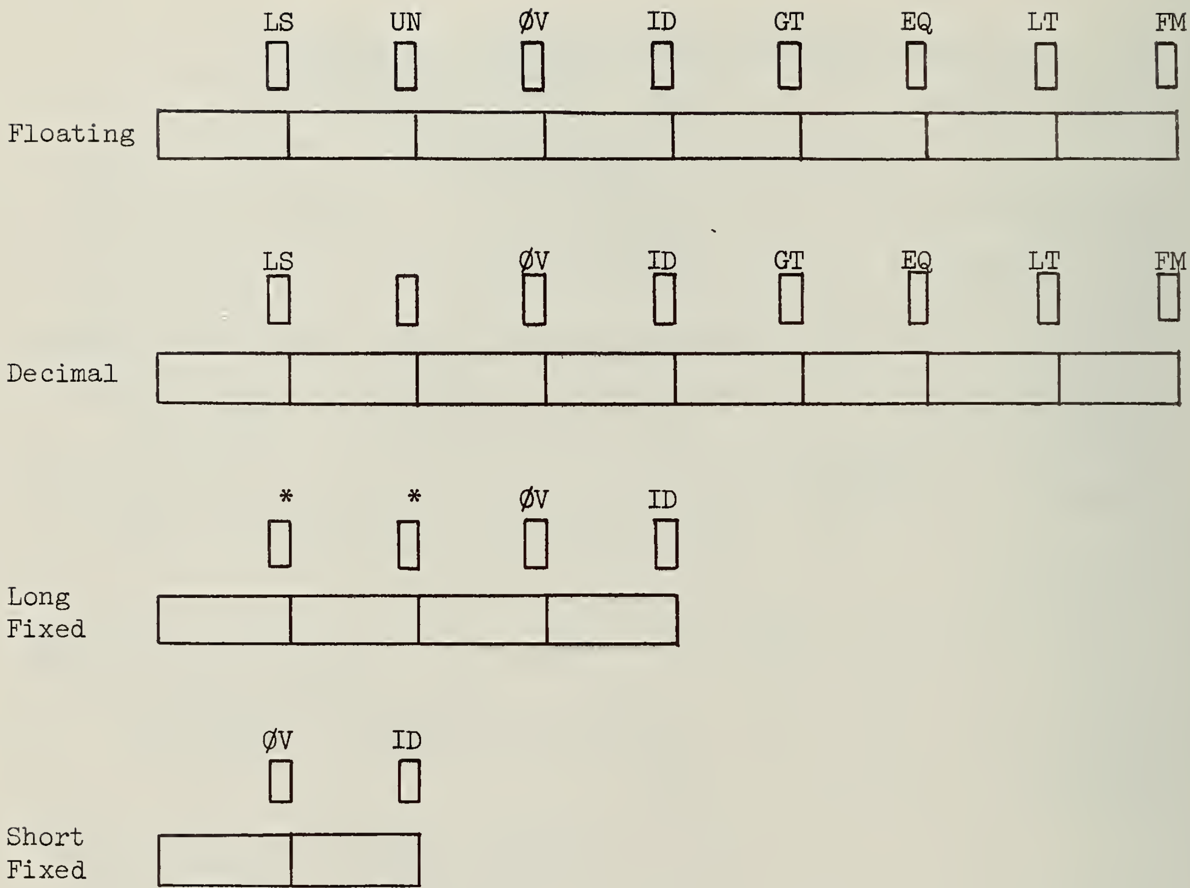
The bogus result with the appropriate computational condition flag(s) set is pushed into the ØS as if it were a correct result. An interrupt then takes place except for an overflow for fixed point multiply. (See Section 2.3.2.3).

The following parts in this section describe the Illiac III computational condition indicators and illustrate the disposition of bogus results. The assignment of computational condition flags is illustrated for the various number types in Figure 2.3.3.1/1. The comparison indicator flags (Section 2.3.2) are also shown for the sake of completeness.

For the reader familiar with PL/1, Table 2.3.3.1/2 describes the analogy between the computational conditions of PL/1* and those implemented in the hardware of Illiac III. Although there is not a one-to-one correspondence between the two versions, both yield approximately the same information if the instruction and number type are known.

---

*"IBM Operating System/360, PL/1 - Language Specifications," File No. S360-29, Form C28-6571-4, p. 162.



* = Not Used in this Number Type

Figure 2.3.3.1/1 Flag Bit Designation for Arithmetic Indicators

Table 2.3.3.1/2 Correspondence between Computational Conditions of PL/1 and Those of Illiac III

PL/I	ILLIAC III		
	Instruction Variant	Number Type	Computational Condition Indicator
CONVERSION	CVF or CVL	Decimal	Invalid Decimal Data
	CVD or CVL	Floating	Overflow
	CVL	Decimal	Overflow
FIXEDOVERFLOW	ADD, SUB, ABS, MPY, DIV, POLY	Long or Short Fixed or Decimal	Overflow
OVERFLOW	ADD, SUB, MPY, DIV, POLY,	Floating	Overflow
SIZE	---	---	No hardware implementation.
UNDERFLOW	ADD, SUB, MPY DIV, POLY	Floating	Underflow
ZERODIVIDE	DIV	Any	Overflow

2.3.3.2 Overflow (OV)

Table 2.3.3.2 - Overflow (OV)

Order In Which Condition May Occur	Description of Condition	Result In Stack
ADD, SUB		
S. Fixed (also in CPRA)	Magnitude of result exceeds $(2^{15}-1)$.	Low order 16 bits of the result.
L. Fixed (also in CPRA)	Magnitude of result exceeds $(2^{31}-1)$.	Low order 32 bits of the result.
Floating	The exponent of the normalized result exceeds 63 and the result fraction is not zero.	Fraction is that computed and correctly normalized. Sign of fraction is correct. Exponent = +63.
Decimal	Magnitude of result exceeds 99999999999999 (14, 9's).	Low order 14 digits of the result.
MPY, POLY		
S. Fixed	Magnitude of result exceeds $(2^{15}-1)$.	Full-word correct result with OV flag set.

Order in Which Condition May Occur	Description of Condition	Result in Stack
L. Fixed	Magnitude of result exceeds $(2^{31}-1)$.	Double word (integer) correct result with OV flag set.
Floating	The exponent of the normalized result exceeds 63 and the result fraction is not zero.	Fraction is that computed and correctly normalized. Sign of fraction is correct. Exponent = +63.
Decimal	Magnitude of result exceeds 14, 9's.	Low order 14 digits of the result.
DIV		
S. Fixed L. Fixed	Division by zero.	Largest number representable. Sign of result is sign of dividend. Remainder is 0.
Floating	Division by zero.	Fraction and exponent are all 1's. Sign of fraction is the sign of the dividend.
Decimal	Division by zero.	Largest number representable.

Table 2.3.3.2 - Overflow (OV) (Continued)

Order in Which Condition May Occur	Description of Condition	Result in Stack
ABS, NEG		
S. Fixed	Attempt to negate the most negative number representable.	The integer +1.
CVD		
Floating	The magnitude of the converted number exceeds the range of the new number type.	Result is the converted * number with missing high order digits which have overflowed.
CVL		
Floating Decimal	The magnitude of the converted number exceeds the range of the new number type.	Result is the converted number with missing high order bits which have overflowed.

* This is true only if Exponent of Floating Operand is $\leq$ 14. Otherwise "double" overflow occurs and error is compounded.

2.3.3.3 Underflow (UN)

Table 2.3.3.3 - Underflow (UN)

Order In Which Condition May Occur	Description of Condition	Result in Stack
ADD SUB MPY DIV POLY Floating Only	The exponent of the normalized result is less than -64 and the result fraction is not zero.	Fraction is that computed and correctly normalized. Sign of fraction is correct. Exponent is -64.

2.3.3.4 Invalid Decimal Data (ID)

Table 2.3.3.4 - Invalid Decimal Data (ID)

Order In Which Condition May Occur	Description of Condition	Result in Stack
Any Decimal Order Except NEG, ABS, MNS, TA.	A sign or digit code of an operand is incorrect.	Result is the contents of the AU accumulator when the decoding error was detected. No ID check is made for unary operations except CVF and CVL.

2.3.3.5 Loss of Significance (LS)

Table 2.3.3.5 - Loss of Significance (LS)

Order In Which Condition May Occur	Description of Condition	Result in Stack
ADD SUB POLY Floating Only	Fraction of result is 0. Exponent of result $\neq -64$. For example-when two equal numbers are subtracted.	True zero with LS flag set.

2.4 Instructions Executed by the Pattern Articulation Unit

The PAU instructions have been divided into classes defined by the number of explicit operand planes required for the execution of a given instruction. All instructions pertaining to loading and unloading the Iterative Array (i.e. border instructions) have been included in a separate section.

2.4.1 Conventions

2.4.1.1 Planes and Borders

The word plane as used to describe PAU instructions, refers exclusively to the 32 x 32 array. Any manipulations of the borders will be explicitly stated. The symbols S_1, ..., S_n are used to refer to one or more elements of the set M, P0, ..., P55 of planes.

If any restrictions on plane addresses exist they will be stated explicitly.

Although the M-plane (and its borders) is used by the PAU as a buffer and scratch plane in the execution of many instructions, the programmer may use the M-plane as an operand plane. It has been explicitly stated whether the M-plane and/or its borders are altered during the execution of an instruction.

The numbering convention shown in Figure 2.4.1/1 is used explicitly by the PAU control and is used in dumps and thresholding operations. The PAU treats the plane address as a mod 64 number with -1 (the M-plane) represented as 111111. The ten high order bits of the (halfword) plane address are ignored.

2.4.1.2 Direction Numbers

Direction numbers are used to specify the relative location of neighbors of a given cell. Thus the direction number assigned to a cell is dependent upon the current topology of the array. (See Figure 2.4.1/2).

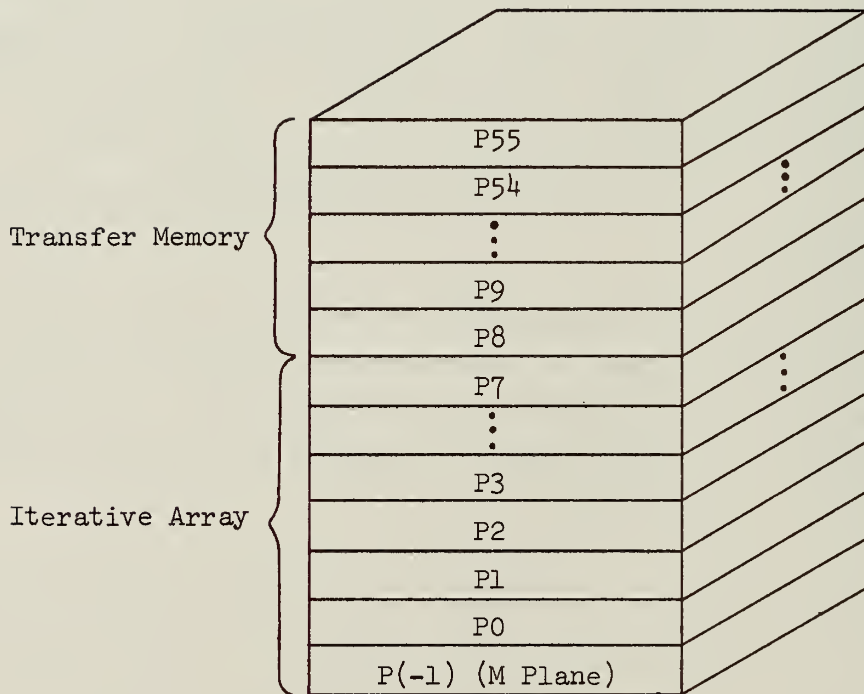
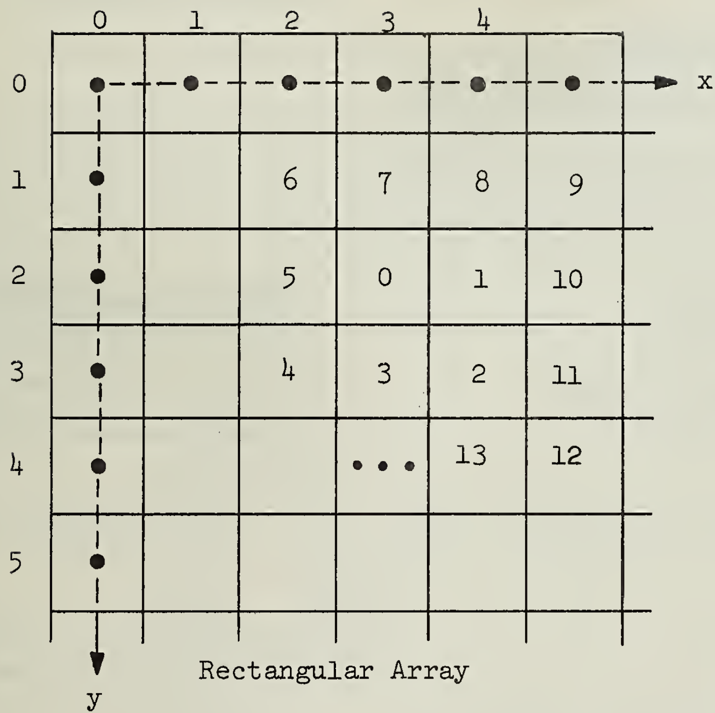


Figure 2.4.1/1 PAU Plane Numbering Convention



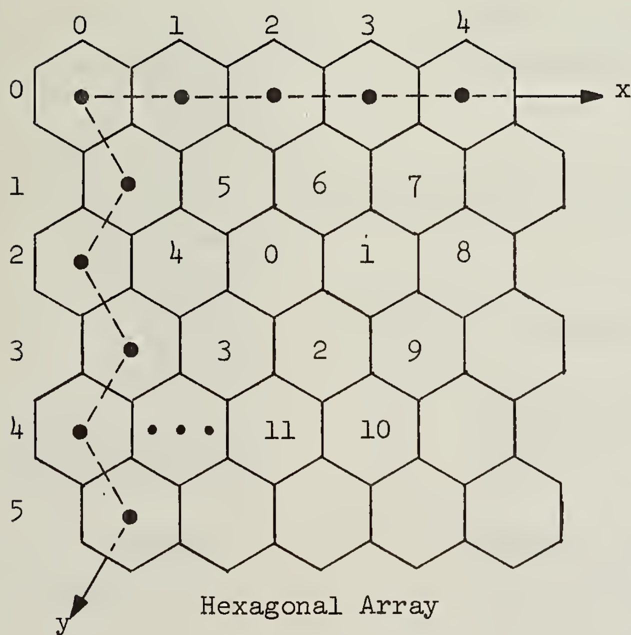


Figure 2.4.1/2 Direction Numbers as Functions of the Topologies of the Iterative Array

2.4.1.3 Indicators

2.4.1.3.1 Exceptional Conditions

The following indicators are used by the PAU to signal the occurrence of an error during the execution of a PAU instruction.

Indicator	Cause for Occurrence
Parity (UPE)	Incorrect parity detected
Malfunction (UM)	Detected hardware error
TM OFF (TMO)	Transfer memory accessed while off-line
Invalid Plane Address (IPA)	1) Plane address (6 least significant bits) not in range - 1 to 55 2) Address >7 specified for instructions valid only in IA
Count Overflow (CNTOV)	Count greater than 1024 specified
Coordinate Overflow (CORDOV)	Coordinate >4095 specified in plot instructions
TM-TM Request (TMTMRQ)	Invalid specification for COPYC
Invalid Op Code (IOC)	Attempted execution of an instruction which is not a PAU instruction or illegal instruction after interrupt
Instruction Not Present (INP)	Attempted execution of an unassigned or unavailable PAU instruction
Invalid Direction List (IDL)	Neighbors 7 and 8 specified in HEX mode
BOOLE Flag 1 (BF1)	No variables in elementary function
BOOLE Flag 2 (BF2)	Variables defined as both true and complemented
BOOLE Flag 3 (BF3)	Forbidden OP(s) code
BOOLE Flag 4 (BF4)	Insufficient Stack Depth

2.4.1.3.2 Indicator Halfword

Upon the recognition of an exceptional condition the PAU will return a BOGUS RESULT indicator to the calling TP as well as placing a halfword of indicators on the XN OUTBUS. The TP will store these indicators in the interrupt segment. This is the only information returned to the TP (i.e. no bogus data will be returned).

The OUTBUS assignments are:

Bit:		
	10	UPE
	11	UM
	12	TMO
	13	IPA
	14	CNTOV
	15	CORDOV
	16	TMTMRQ
	17	
	18	-
	20	INP
	21	IOC
	22	IDL
	23	BF1
	24	BF2
	25	BF3
	26	BF4
	27	
	28	-

2.4.2 Zero-Plane Instructions

The instructions in this section have no planar operands. They are used to set up conditions of operation for the Iterative Array.

The last two instructions (RESUME and RESTART) are privileged instructions initiated by the supervisor after the PAU has been interrupted. One of these two instructions must be executed after the interrupt is serviced or the IOC flag will be set.

2.4.2.1 Topology

TOPOLOGY [1 1|0 1 1 1 S C] [1]

This is a declarative instruction which sets the topology of interconnection of the PAU. The topology remains the same until explicitly changed with another Topology instruction.

The variants are Rectangular/Hexagonal and Planar/Toroidal. Rectangular/Hexagonal refers to the two types of cellular array which can be manipulated. Rectangular means the 4-sided (8-nearest neighbors) configuration; Hexagonal means the 6-sided (6-nearest neighbors) connection. Planar/Toroidal refers to connections which affect the Shift order explicitly and other orders implicitly.

If Planar is specified, bits shifted out of the borders are lost and zeros are entered at the opposite edge.

If Toroidal is specified, bits which are shifted out of the borders "wrap around" and are entered at the opposite edge.

The default options are Rectangular and Planar.

S	C	Topology
0	0	Rectangular/Planar
0	1	Rectangular/Toroidal
1	0	Hexagonal/Planar
1	1	Hexagonal/Toroidal

Indicators: Parity

2.4.2.2 **Set Origin**

SETØRG | 1 1 | 0 0 0 0 1 1 | [1] | <Operand>$_S$

Set Iterative Array Origin: Set the origin (0,0) of the Iterative Array to the value specified by the (X,Y) coordinate in the same format as used in the scanner coordinate mode. The coordinates must lie in the range $0 \leq XORG, YORG \leq 4095$. (The three low order bits of the 15 bit coordinates (the SMV vernier bits) are ignored.) The origin remains the same until explicitly changed by another SETØRG command.

Indicators: Parity

2.4.2.3 Resume

RESUME | 1 1 | 0 0 0 0 0 1 | [1]

Resume PAU operation: This instruction signals the PAU to resume normal operation after an interrupt. The PAU will complete execution of the interrupted instruction or, if already completed, will accept a new instruction. All status indicators retain the state they were in at interruption. This is a protected instruction.

Indicators: Parity

2.4.2.4 Restart

RESTART | 1 1 | 0 0 0 0 0 0 | [1]

Restart PAU operation: This instruction signals the PAU to reset all status indicators to the default (initialized) condition and to accept a new instruction. This is a protected instruction.

Indicators: Parity

2.4.3 One-Plane Instructions

The one-plane instructions may be divided into two types. The first type uses only the Iterative Array for execution. The second type uses both the Iterative Array and the Transfer Memory. Instructions of this latter type are of the LIST and PLOT variety.

Note that in the explanations of the LIST and PLOT type instructions that the origin of the Iterative Array is given as (x,y) = (0,0). The programmer may assign a virtual origin by means of the zero-plane instruction SETORG. The coordinates then PLOTTED or LISTED will be the virtual coordinates i.e X, Y = XORG + x, YORG + y where XORG, YORG is defined by a SETORG, and x, y ($0 \leq x,y \leq 31$) are the relative Iterative Array coordinates.

Furthermore, in the PLOT instructions a coordinate out of the range of the virtual Iterative Array will be ignored unless it is greater than 4095 in which case an error is indicated.

2.4.3.1 Data Formats

2.4.3.1.1 Coordinate Mode

The coordinate mode used by the PAU is similar to that put out by the scanner. The format is as follows:

$Y[1]^{1}X[1,1]$ A[1,1] ... X[1,m] A[1,m]

$Y[2]^{1}X[2,1]$ A[2,1] ... X[2,n] A[2,n]

⋮

$Y[K]^{1}X[K,1]$ A[K,1] ... X[K,q] A[K,q]

Y[K+1] (both flags 1)

where

X,Y - Two bytes each, $0 \leq X \leq 4095$, $0 \leq Y \leq 4095$. Right justified

A - Attribute - either 0 bits, 8 bits (1 byte right justified in a halfword), or 48 bits (6 bytes)

2.4.3.1.2 Incremental Code

±	Δx			±	Δy			8
0	1	2	3	4	5	6	7	

Bit 0 Sign of x displacement

Bits 1-3 Magnitude of x displacement ($0 \leq x \leq 7$)

Bit 4 Sign of y displacement

Bits 5-7 Magnitude of y displacement ($0 \leq y \leq 7$)

2.4.3.2 Clearp

CLEARP | 1 1 | 1 1 0 0 0 0 | [1] <ID Byte> <Operand S_1>

Clear plane and border: Set the contents of plane S_1 and/or its borders, as specified by the Identification byte, to zero. <u>S_1 must be in the IA.</u>

ID Byte | P | E | SE | S | SW | W | NW | N | [NE]

where P = Plane
E = East Border
SE = Southeast Corner
S = South Border
SW = Southwest Corner
W = West Border
NW = Northwest Corner
N = North Border
NE = Northeast Corner

Indicators: Parity, Invalid Plane Address

2.4.3.3 Setp

SETP | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | [1] <ID Byte> <Operand S_1>

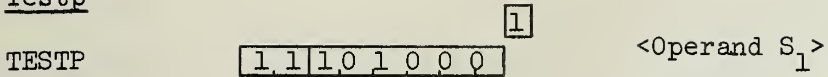
Set plane and border: Set the contents of plane S_1 and/or its borders, as specified by the Identification byte, to one. S_1 must be in the IA.

ID Byte | P | E | SE | S | SW | W | NW | N | [NE]

where P = Plane
E = East Border
SE = Southeast Corner
S = South Border
SW = Southwest Corner
W = West Border
NW = Northwest Corner
N = North Border
NE = Northeast Corner

Indicators: Parity, Invalid Plane Address

2.4.3.4 Testp

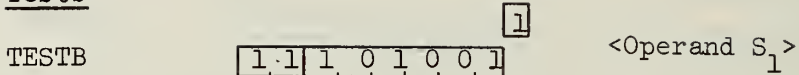


Test plane: Plane S_1 is copied into the M-plane. The M-plane is then tested for zero. If the M-plane is identically zero, the EQ indicator in the TP will be turned on (set to one). If the M-plane is not identically zero, the EQ indicator will be turned off (set to zero).

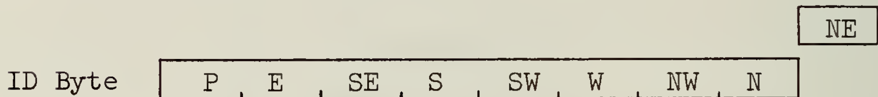
The previous contents of the M-plane will be destroyed.

Indicators: Parity, Invalid Plane Address.

2.4.3.5 Testb

TESTB  <Operand S_1>

Test Borders: The borders of the plane S_1 are tested for zero. S_1 must be in the IA.
If all borders are identically zero, the EQ indicator in the TP will be turned on (set to one).
If all borders are not identically zero, the EQ indicator will be turned off (set to zero). Also one byte (right-justified in a half-word) in the same format as the Identification Byte indicating (by setting appropriate bits to '1') which borders are non-zero will be pushed into the Operand Stack.

ID Byte 

where P = Plane
E = East Border
SE = Southeast Corner
S = South Border
SW = Southwest Corner
W = West Border
NW = Northwest Corner
N = North Border
NE = Northeast Corner

Indicators: Parity, Invalid Plane Address

2.4.3.6 Replicate

REPLICATE | 1 1 | 1 0 0 1 0 0 | [1] | <Operand S_1>

The contents of plane S_1 (including borders) is duplicated in each plane of the IA. S_1 must be in the IA.

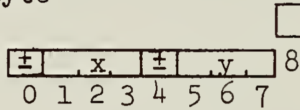
Indicators: Parity, Invalid Plane Address.

2.4.3.7 Shift

SHIFT <Operand S_1> <Operand D>

Shift plane: Shift plane S_1 and its borders as specified by the displacement byte, D.

Displacement Byte



Bit 0	Sign of **x** displacement ($\pm$)
Bits 1-3	Magnitude of **x** displacement ($0 \leq x \leq 7$)
Bit 4	Sign of y displacement ($\pm$)
Bits 5-7	Magnitude of y displacement ($0 \leq y \leq 7$)

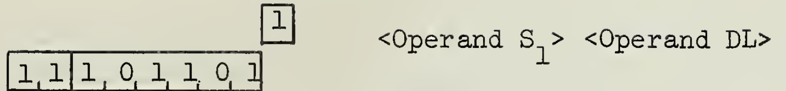
S_1 must be in the IA and may not be in the M-plane.

The previous contents of the M-plane will be destroyed.

Indicators: Parity, Invlaid Plane Address.

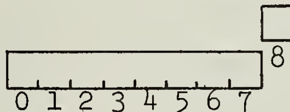
2.4.3.8 Tally

TALLY <Operand S_1> <Operand DL>



Tally ones up: Set $P(i) = P(S_1)\&P(i-1) \mid P(i), 0 \leq i \leq 7$. Effectively $P(-1)=1$.

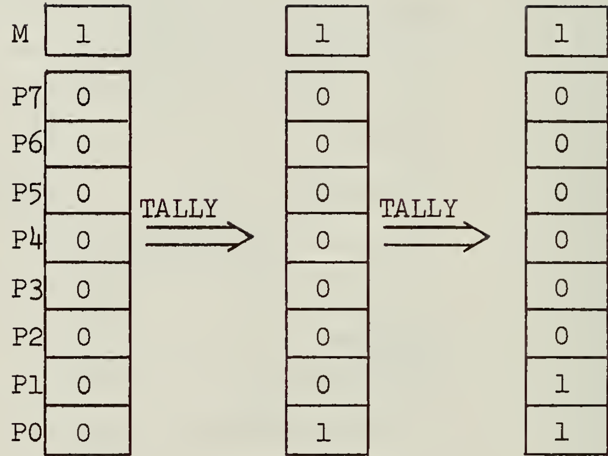
S_1 is copied into the M-plane. The value of the M-plane replaces the first zero over every ascending vertical seqeunce of ones. (See examples given below.) The M-plane will contain S_1 after execution. The contents of P0-P7 may be changed. No check for overflow will be made. The direction list, DL, provides for tallying nearest neighbors without shifting. Sequential TALLY's will be performed on those nearest neighbors specified in the direction list. A direction list consists of one byte in the following format.



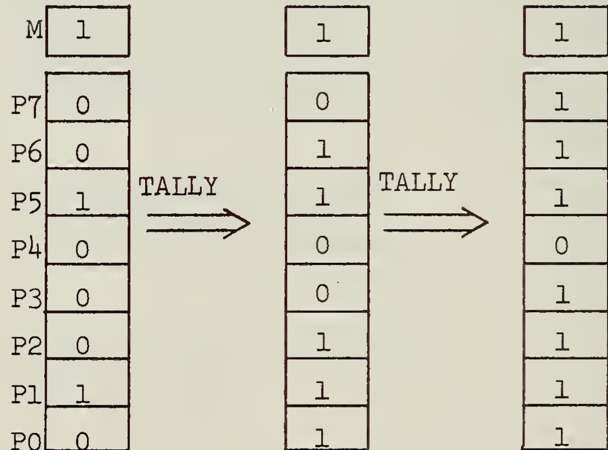
Where the appropriate bit set to one indicates that neighbor is to be used in the execution of the instruction.

Indicators: Parity, Invalid Plane Address, Invalid Direction List.

Example 1:



Example 2:



2.4.3.9 Tallyho

TALLYHO [1 1 | 1 0 1 1 1 0] [1] <Operand S_1> <Operand DL>

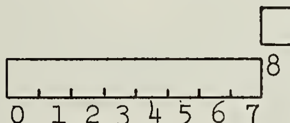
Tally zeroes down: Set $P(i) = \neg[P(S_1) \& \neg P(i+1)] \& P(i)$, $0 \leq i \leq 7$.

Effectively $P(8) = 0$.

S_1 is copied into the M-plane. The complement of the value of the M-plane replaces the first one under every descending vertical sequence of zeroes. (See examples given below).

The M-plane will contain S_1 after execution. The contents of P0-P7 may be changed. No check for underflow will be made.

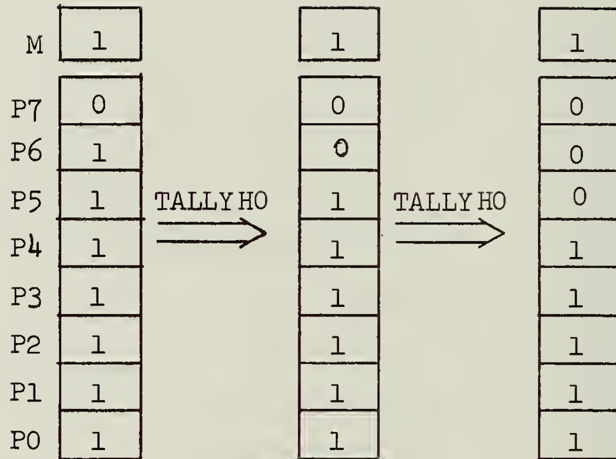
The direction list, DL, provides for tallying nearest neighbors without shifting. Sequential TALLYHØ's will be performed on those nearest neighbors specified in the direction list. A direction list consists of one byte in the following format.



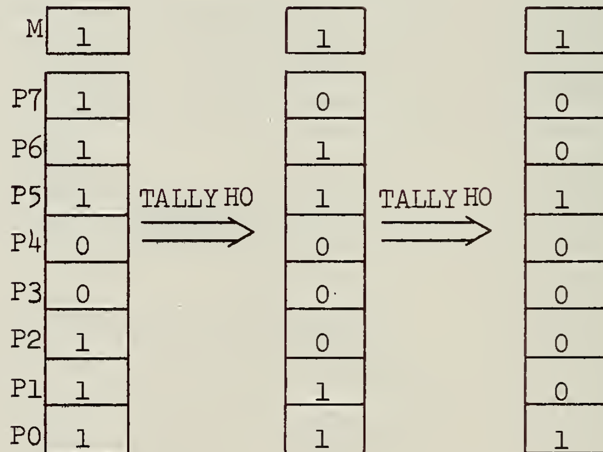
Where the appropriate bit set to one indicates that neighbor is to be used in the execution of the instruction.

Indicators: Parity, Invalid Plane Address, Invalid Direction List.

Example 1:



Example 2:



2.4.3.10 <u>Area</u>

AREA | 1 1 | 1 0 0 0 0 0 0 | [1] <Operand S_1> <Count n>

Count ones: Count the number of ones in S_1 and return the count to the Operand Stack as a halfword integer under the following conditions:

If S_1 contains not more than <u>n</u> one bits, then all such points in S_1 are counted, and the count, C, is placed in the OS.

If S_1 contains more than <u>n</u> one bits, the first n points are counted and n+1 is placed in the OS.

<u>Important</u>: Counted points are erased (set to zero) in the operand plane. A sequential TV-type scan of S_1 is initiated starting at (0,0) scanning in the positive x direction. Ones encountered in the scan are counted.

S_1 is copied into the M-plane. The ones of the M-plane are listed and replaced by zeros. At the end of execution, the M-plane is copied into S_1.

Note: S_1 must be in the Iterative Array and may not be the M-plane.

Indicators: Parity, Invalid Plane Address, Count OV.

2.4.3.11 List

LIST [1 1|1 0 0 0 0 1] [1] <Operand S_1> <Count n>

List plane: A sequential TV-type scan of S_1 is initiated starting at (x,y) = (0,0) scanning in the positive x direction. The coordinate pairs (X,Y) of one bits are sequentially stored in the Operand Stack in the same format used in the scanner coordinate mode. The angular coordinate, θ, is not allowed. See Section 2.4.3.1.1.

If S_1 contains not more than n one bits, then all such points in S_1 are listed, and the count, C, is placed in the Operand Stack.

If S_1 contains more than n one bits, the first n points are listed and n+1 is placed in the Operand Stack. Important: The listed points are erased (set to zero) in the operand plane.

S_1 is copied into the M-plane. The ones of the M-plane are listed and replaced by zeros. At the end of execution the M-plane is copied into S_1. Note: S_1 must be in the Iterative Array and may not be the M-plane.

Indicators: Parity, Invalid Plane Address, Count OV.

2.4.3.12 Listsz

LISTSZ [1] [1 1 | 1 0 0 0 1 0] <Operand S_1> <Count n>

List plane and read short z-words: A sequential TV-type scan of S_1 is initiated starting at (x,y) = (0,0) scanning in the positive x direction. The coordinate pairs (X,Y) of one bits are sequentially stored in the Operand Stack in the same format used in the scanner coordinate mode. The angular coordinate, θ, is not allowed. See Section 2.4.3.1.1.

Following each coordinate pair the associated 8-bit (1-byte) z-word contained in planes P8-P15 (Transfer Memory) is listed right-justified in a halfword.

If S_1 contains not more than _n_ one bits, then all such points in S_1 are listed, and the count, C, is placed in the Operand Stack.

If S_1 contains more than _n_ one bits, the first n points are listed and n+1 is placed in the Operand stack. Important: The listed points are erased (set to zero) in the operand plane.

S_1 is copied into the M-plane. The ones of the M-plane are listed and replaced by zeros. At the end of execution the M-plane is copied into plane S_1.
Note: S_1 must be in the Iterative Array and may not be the M-plane.

Indicators: Parity, Invalid Plane Address, Count OV.

2.4.3.13 Listlz

LISTLZ | 1 1 | 1 0 0 0 1 1 | [1]   <Operand S_1> <Count n>

List plane and read long z-words: A sequential TV-type scan of S_1 is initiated starting at (x,y) = (0,0) scanning in the positive x direction. The coordinate pairs (X,Y) of one bits are sequentially stored in the Operand Stack in the same format used in the scanner coordinate mode. The angular coordinate, θ, is not allowed. See Section 2.4.3.1.1.

Following each coordinate pair the associated 48-bit (6 byte) z-word contained in planes P8-P55 (Transfer Memory) is listed.

If S_1 contains not more than n one bits, then all such points in S_1 are listed, and the count, C, is placed in the Operand Stack.

If S_1 contains more than n one bits, the first n points are listed and n+1 is placed in the Operand Stack.
Important: The listed points are erased (set to zero) in the operand plane.

S_1 is copied into the M-plane. The ones of the M-plane are listed and replaced by zeros. At the end of execution the M-plane is copied into plane S_1.
Note: S_1 must be in the Iterative Array and may not be the M-plane.

Indicators: Parity, Invalid Plane Address, Count OV.

2.4.3.14 Listi

LISTI [1] | 1 1 | 1 0 1 0 1 0 | <Operand S_1>

List plane incremental: A sequential TV-type scan of S_1 is initiated starting at (x,y)=(0,0) scanning in the positive x-direction until a one bit is encountered.

The coordinate pair of this point is returned to the operand stack as two halfwords in the order Y, X.

Following the coordinate pair is an incremental string corresponding to a contour trace of the figure. See Section 2.4.3.1.2. The algorithm used may be found in Seitz, C.L., "An Opaque Scanner for Reading Machine Research" (MIT thesis).

At the completion of the tracing algorithm the figure is deleted.

S_1 must be in the Iterative Array and may not be the M-plane.

Indicators: Parity, Invalid Plane Address.

2.4.3.15 Readlz

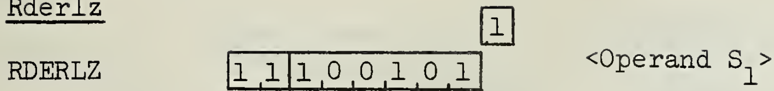
READLZ [1] [1,1|1,1,0,0,1,1]

Read long z-word: The 48 bit (6 byte) z-word at the point specified by the coordinate pair in the word at the top of the operand stack is pushed into the operand stack.

The coordinate pair consists of two halfwords; the first being the Y-coordinate, the second being the -X. The pair is popped and the long z-word pushed into the stack. If the point is in the virtual Iterative Array the flags will be set to zero. If the point is not in the virtual Iterative Array a zero z-word with the flags set to one will be pushed.

Indicators: Parity, Coordinate OV.

2.4.3.16 Rderlz



Read long z-word and erase point: The 48 bit (6 byte) z-word at the point specified by the coordinate pair in the word at the top of the operand stack is pushed into the operand stack. If the specified point (in S_1 is a one, it is erased (set to zero). Otherwise it is unchanged.

The coordinate pair consists of two halfwords: the first being the Y-coordinate, the second being the X. The pair is popped and the long z-word pushed into the stack. If the point is in the virtual Iterative Array the flags will be set to zero. If the point is not in the virtual Iterative Array a zero z-word with the flags set to one will be pushed.

S_1 must be in the Iterative Array and may not be the M-plane.

Indicators: Parity, Invalid Plane Address, Coordinate OV.

2.4.3.17 <u>Erasep</u>

ERASEP [1 1 1 0 1 0 1 1] [1] <Operand S_1>

Erase point: A sequential TV-type scan of S_1 is initiated starting at (x,y) = (0,0) scanning in the positive x-direction until a one bit is encountered. The one bit is then erased (set to zero).

<u>Note</u>: S_1 must be in the Iterative Array and may not be the M-plane.

Indicators: Parity, Invalid Plane Address.

2.4.3.18 Plot

PLOT [1 1|1 1 0 1 0 0] [1] <Operand S_1> <Operand>

Plot plane: A string of points is plotted (set to 1) in S_1. The string of (X,Y) coordinates, specified by the operand, is in the same format used in the scanner coordinate mode. The angular coordinate, θ, is not allowed. See Section 2.4.3.1.1.

Note: $0 \leq X \leq 4095$, $0 \leq Y \leq 4095$.

Note: S_1 must be in the Iterative Array and may not be the M-plane. The contents of the M-plane will be destroyed.

Indicators: Parity, Invalid Plane Address, Coordinate OV (Plot out of bounds).

2.4.3.19 Plotsz

PLØTSZ | 1 1 | 1 1 0 1 0 1 | [1] | <Operand S_1> <Operand>

Plot plane and write short z-word: A string of points is plotted (set to one) in S_1. The short z-word (8-bits) associated with each coordinate pair is written into planes P8-P15 (Transfer Memory) at the plotted coordinate. The string of (X,Y) coordinates, specified by the operand is in the same form as the scanner coordinate mode. The angular coordinate, θ, is not allowed. See Section 2.4.3.1.1.

Note: $0 \leq X \leq 4095$, $0 \leq Y \leq 4095$.

The contents of the M-plane and the previous contents of planes P8-P15 (at the plotted coordinates) will be destroyed.

Note: S_1 must be in the Iterative Array and may not be the M-plane.

Indicators: Parity, Invalid Plane Address, Coordinate OV (Plot out of bounds).

2.4.3.20 Plotlz

PLØTLZ [1 1|1 1 0 1 1 0] [1] <Operand S_1> <Operand>

Plot plane and write long z-word: A string of points is plotted (set to one) in S_1. The long z-word (48-bits) associated with each coordinate pair is written into planes P8-P55 (Transfer Memory) at the plotted coordinate. The string of (X,Y) coordinates is in the same form as the scanner coordinate mode. The angular coordinate, θ, is not allowed. See Section 2.4.3.1.1.

Note: $0 \leq X \leq 4095$, $0 \leq Y \leq 4095$.

The contents of the M-plane and the previous contents of planes P8-P55 (at the plotted coordinates) will be destroyed.

Note: S_1 must be in the Iterative Array and may not be the M-plane.

Indicators: Parity, Invalid Plane Address, Coordinate OV (Plot out of bounds).

2.4.3.21 Ploti

PLØTI | 1 1 | 1 1 0 1 1 1 | [1] | <Operand S_1> <Operand>

Plot plane incremental: A string of points is plotted (set to one) in S_1. The initial (X,Y) coordinate is in the same format used in the scanner incremental mode. Subsequent bytes are interpreted according to the incremental code. (See Section 2.4.3.1.2)

Note: S_1 must be in the Iterative Array and may not be the M-plane. The contents of the M-plane will be destroyed.

Indicators: Parity, Invalid Plane Address, Coordinate OV (Plot out of bounds).

2.4.3.22 <u>Writlz</u>

WRITLZ　　[1 1|1 0 0 1 1 0] [1]　　<Operand S_1>

Write long z-word: A sequential TV-type scan of S_1 is initiated starting at (x,y)=(0,0) scanning in the positive x-direction until a one bit is encountered. The 48-bit (6 byte) z-word at the top of the ØS is popped and then written into P8-P55 (Transfer Memory) at the coordinate position of the one bit.

Note: S_1 must be in the Iterative Array and may not be the M-plane.

Indicators: Parity, Invalid Plane Address.

2.4.3.23 Wrerlz

WRERLZ [1 1|1 0 0 1 1 1] [1] <Operand S_1>

Write long z-word and erase point: A sequential TV-type scan of S_1 is initiated starting at (x,y) = (0,0) scanning in the positive x direction until a one bit is encountered. The 48 bit (6 byte) z-word at the top of the ØS is popped and then written into P8-P55 (Transfer Memory) at the coordinate position of the one bit. The one bit is then erased (set to zero).

Note: S_1 must be in the Iterative Array and may not be the M-plane.

Indicators: Parity, Invalid Plane Address.

2.4.4 Two-Plane Instructions

The two-plane instructions may be divided into two types. The first two instructions are copy instructions useful in moving planes around. The remainder of the two-plane instructions are logical operations on planes. No planar complementation is provided as this function is the same as the copy complement with the same plane specified for both operands.

2.4.4.1 Copy

COPY [1 1 | 1 1 1 0 0 0] [1] <Operand S_1><Operand S_2>

S_2 is copied into S_1. The contents of S_2 remain unchanged.

The disposition of the borders and contents of the M-plane after execution are as follows:

Loc(S_1)	Loc(S_2)	Disposition of Borders	C(M)
IA	TM	Clear	S_2
TM	IA	Drop	S_2
IA	IA	Copy	Unchanged
TM	TM	No borders	S_2

Indicators: Parity, Invalid Plane Address.

2.4.4.2 Copyc

COPYC ⎡1 1⎤1 1 1 0 0 1⎤ ⎡1⎤ <Operand S_1><Operand S_2>

The complement of S_2 is copied into S_1. The contents of S_2 remain unchanged. Note that a COPYC between two planes in the TM (P8-P55) is not permitted.

The disposition of the borders and contents of the M-plane after execution are as follows:

Loc(S_1)	Loc(S_2)	Disposition of Borders	C(M)
IA	TM	Clear	S_2
TM	IA	Drop	$\overline{S_2}$
IA	IA	Copy Complement	Unchanged
TM	TM	Disallowed	Disallowed

Indicators: Parity, Invalid Plane Address, TM-TM Request

2.4.4.3 Pland

PLAND   [1]
[1 1 | 1 1 1 0 1 0]   <Operand S_1><Operand S_2>

Planar AND: The bitwise AND of planes S_1 and S_2 (and their borders) replace the contents of plane S_1. The contents of plane S_2 remain unchanged. S_1 and S_2 must be in the Iterative Array and may not be the M-plane.

Indicators: Parity, Invalid Plane Address.

2.4.4.4 <u>Plor</u>

PLOR <Operand S_1><Operand S_2>

Planar OR: The bitwise OR of planes S_1 and S_2 (and their borders) replace the contents of plane S_1. The contents of plane S_2 remain unchanged. S_1 and S_2 must be in the Iterative Array and may not be the M-plane.

Indicators: Parity, Invalid Plane Address.

2.4.4.5 Plnand

PLNAND | 1 1 | 1 1 1 1 0 0 | [1] | <Operand S_1><Operand S_2>

Planar NAND: The bitwise NAND of planes S_1 and S_2 (and their borders) replace the contents of plane S_1. The contents of plane S_2 remain unchanged. S_1 and S_2 must be in the Iterative Array and may not be the M-plane.

Indicators: Parity, Invalid Plane Address

2.4.4.6 Plnor

PLNOR	1 1 | 1 1 1 1 0 1 | [1]	<Operand S_1><Operand S_2>

Planar NOR: The bitwise NOR of planes S_1 and S_2 (and their borders) replace the contents of plane S_1. The contents of plane S_2 remain unchanged. S_1 and S_2 must be in the Iterative Array and may not be the M-plane.

Indicators: Parity, Invalid Plane Address.

2.4.4.7 Plexor

PLEXOR [1] [1 1 | 1 1 1 1 1 0] <Operand S_1> <Operand S_2>

Planar EXCLUSIVE OR: The bitwise EXCLUSIVE OR of planes S_1 and S_2 (and their borders) replace the contents of plane S_1. The contents of plane S_2 remain unchanged. S_1 and S_2 must be in the Iterative Array and may not be the M-plane.

Indicators: Parity, Invalid Plane Address

2.4.4.8 Pleqv

PLEQV <Operand S_1> <Operand S_2>

Planar EQUIVALENCE: The bitwise EQUIVALENCE of planes S_1 and S_2 (and their borders) replace the contents of plane S_1. The contents of plane S_2 remain unchanged. S_1 and S_2 must be in the Iterative Array and may not be the M-plane.

Indicators: Parity, Invalid Plane Address

2.4.5 Three-Plane Instructions

2.4.5.1 Connect

CONNECT [1] [1 1|1 0 1 1 1 1] <Operand S_1><Operand S_2> <Operand S_3><Operand DL>

This instruction determines the connectivity of the digitized graph in S_2, the graph plane. The graph must consist of ones on a field of zeros.

Nodes of the graph are placed in S_3, the nodal or context plane. Nodes must be zeros on a field of ones.

For each point set to one in the M-plane (the initiation plane) a signal is allowed to propagate outward provided the corresponding point in S_3 is 1. Propagation continues in the direction(s) specified in the direction list (DL), along the graph plane lines until the propagating signal reaches a point specified as a node by S_3.

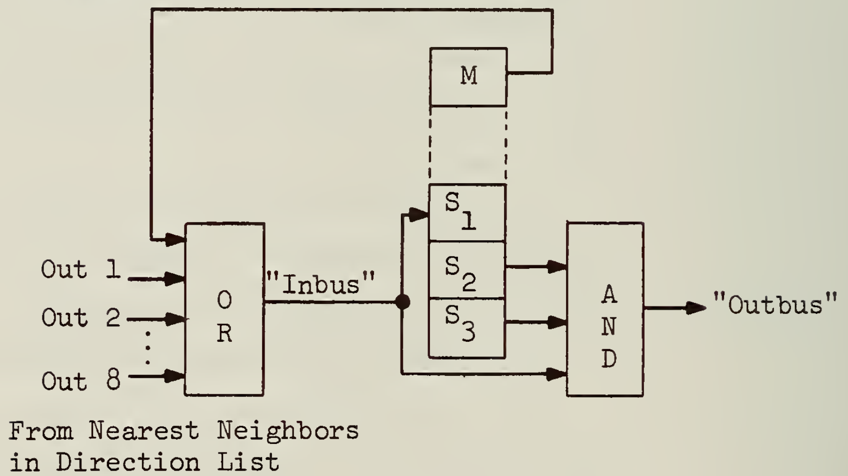
The propagating signal is plotted in S_1. This operation introduces a 'fuzziness' into S_1 since zeros which stop propagation are replaced by ones.

All planes must be in the IA and none can be the M-plane.

The configuration of the Iterative Array during the execution of a CØNNECT instruction is shown in Figure 2.4.5/1. This figure also gives the equations for the propagation phase of the instruction.

Indicators: Parity, Invalid Plane Address.

Logic Connection During
Graphic Search (Propagation Phase)



Propagation Phase

$$\prod_{\substack{\text{Direction}\\ \text{List}}} \text{Out}_i \vee M \rightarrow \text{INBUS} \rightarrow S_1$$

$$\text{INBUS} \;\&\; S_2 \;\&\; S_3 \rightarrow \text{OUTBUS}$$

Figure 2.4.5: Stallactite Configuration During a CONNECT Instruction

2.4.6 Multiple-Plane Instructions

2.4.6.1 Boole

BØØLE   [1 1 0 0 0 0 1 0] [1]   <AL Byte> <Operand>

This instruction allows evaluation of an arbitrary Boolean function of up to 72 (56) cells defined by a cylinder consisting of a cell, its eight (six) nearest neighbors in rectangular (hexagonal) mode and extending these nine (seven) cells to planes $P_0, \ldots, P_n$ where $0 \leq n \leq 7$.

All evaluation of the Boolean function takes place in the Iterative Array. AL is the Availability List and tells what planes in the IA may be used as scratch planes in the evaluation.

The operand phrase is the address of a variable length field which describes in Polish notation the Boolean function to be executed. (See Section 2.4.8 for the description of the syntax and examples).

Indicators: Parity, no variables in elementary functions, variables listed as both true and complemented, forbidden OP(s) code, insufficient stack depth.

2.4.6.2 Gate IA

GATEIA | 1 1 | 0 0 1 0 0 0 | [1]   <IAWØRD word>

This instruction allows the programmer to set the gates of the Iterative Array directly according to the IAWØRD (5 bytes plus flags). See below for the format of the IAWØRD. No check is made to see if the IAWØRD is meaningful. Note that the neighbor numbering will depend on the current topology of the Iterative Array.

(It may prove useful to define pseudo-ops in terms of sequences of these instructions.)

Indicators: Parity.

The IA Word has the following format.

CONTROL BYTE	PLANE READ BYTE	GATE NEIGHBOR BYTE	PLANE WRITE BYTE	IDENTIFICATION BYTE

CONTROL BYTE:

TF	OC	GNM	IC	TØ	TZ	Z/Ø	DØ

TF - True/False Select

OC - Output Complement Select

GNM - Gate Neighbor M (Self M)

IC - Input Complement Select

TØ - Tally One's Select

TZ - Tally Zeroes Select

Z/O - Pulse Zero/One

DØ - Gate Direct Out

PLANE READ BYTE

PRM

PRO	PR1	PR2	PR3	PR4	PR5	PR6	PR7

PRO ≡ Plane Read 0

PR1 ≡ Plane Read 1

PR2 ≡ Plane Read 2

PR3 ≡ Plane Read 3

PR4 ≡ Plane Read 4

PR5 ≡ Plane Read 5

PR6 ≡ Plane Read 6

PR7 ≡ Plane Read 7

PRM ≡ Plane Read M

Gate Neighbor Byte

GN8

GN0	GN1	GN2	GN3	GN4	GN5	GN6	GN7

GATE N0 ≡ Gate Neighbor 0

GATE N1 ≡ Gate Neighbor 1

GATE N2 ≡ Gate Neighbor 2

GATE N3 ≡ Gate Neighbor 3

GATE N4 ≡ Gate Neighbor 4

GATE N5 ≡ Gate Neighbor 5

GATE N6 ≡ Gate Neighbor 6

GATE N7 ≡ Gate Neighbor 7

GATE N8 ≡ Gate Neighbor 8

Plane Write Byte

PWM

PWØ	PW1	PW2	PW3	PW4	PW5	PW6	PW7

PWØ ≡ Plane Write 0

PW1 ≡ Plane Write 1

PW2 ≡ Plane Write 2

PW3 ≡ Plane Write 3

PW4 ≡ Plane Write 4

PW5 ≡ Plane Write 5

PW6 ≡ Plane Write 6

PW7 ≡ Plane Write 7

PWM ≡ Plane Write M

ID Byte

NE

P	E	SE	S	SW	W	NW	N

where P = Plane

E = East Border

SE = Southeast Corner

S = South Border

SW = Southwest Corner

W = West Border

NW = Northwest Corner

N = North Border

NE = Northeast Corner

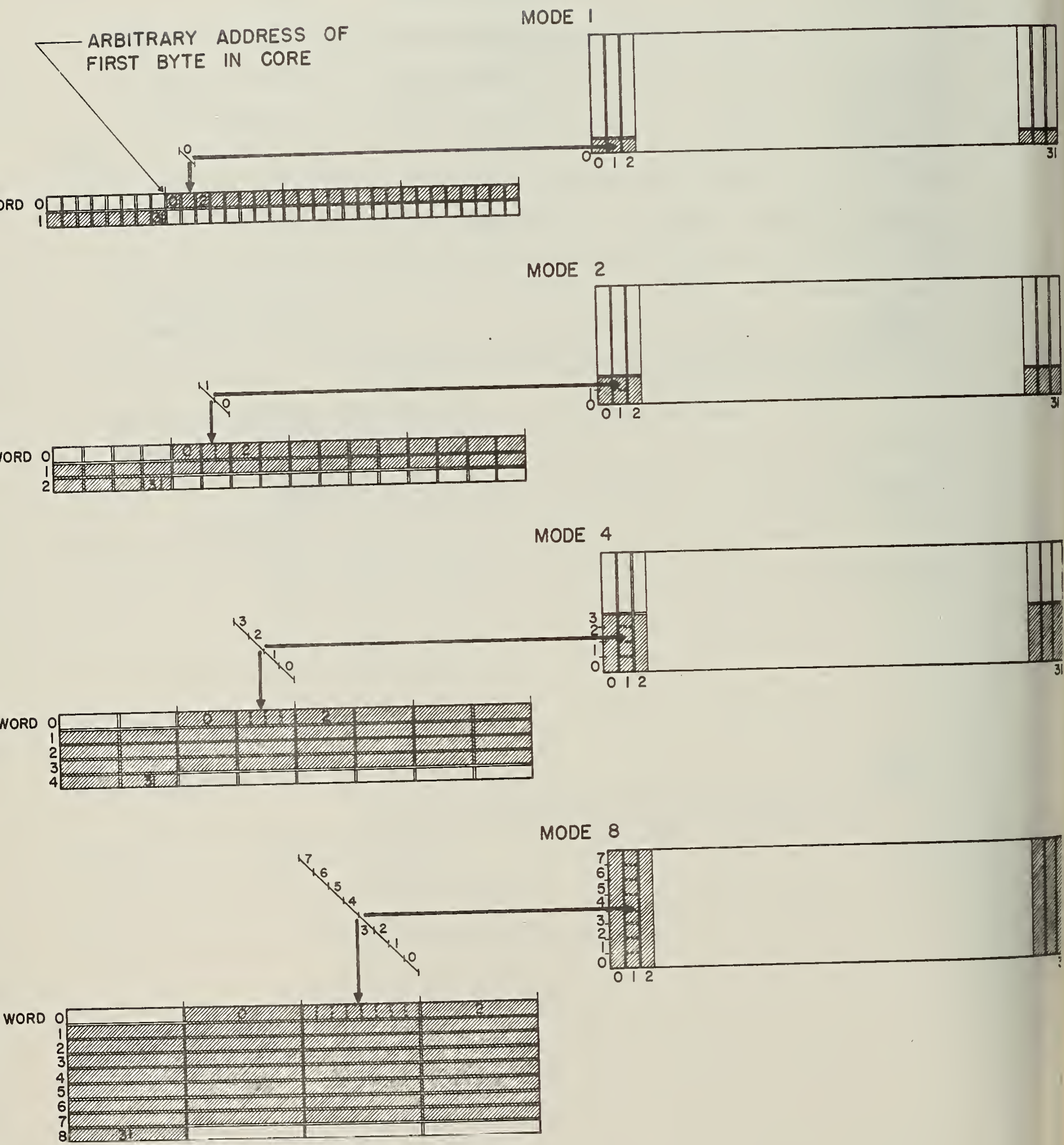
2.4.7 Border Instructions

The following set of instructions are normally used to load and unload the contents of the Iterative Array.

All of the border instructions manipulate data in the same format as the scanner raster mode. Fundamental to the raster mode is the concept of cells. In the simple case, imagine the image divided into many rectangular cells by a piece of screen wire laid over the image. Each cell is assigned a gray level by the scanning unit. The gray scale encoding is under program control and may be either 1, 2, 4, or 8 bits. The result of this encoding is a packed bit string (by rows) of gray level information. Thus if a row has n cells and g bits of gray level encoding, the length of the string will be L = g * n bits. If the end of the row occurs within a byte, the byte is filled out with zeroes.

The Iterative Array may be thought of as a window viewing a 32 by 32 bit portion of the picture and of height equal to the number of planes necessary to represent the gray level encoding.

See Figure 2.4.7/1 for the four cases of loading the interior of the array.



TRANSFER OF DATA BETWEEN CORE AND THE PATTERN ARTICULATION UNIT

2.4.7.1 Data Formats

2.4.7.1.1 Raster String

The raster string used by the PAU has the following format:

$$
\begin{array}{llll}
G_{11} & G_{12} & G_{13} \cdots & G_{1n}{}^{1} \\
G_{21} & G_{22} & \cdots & G_{2n}{}^{1} \\
\vdots & & & \vdots \\
G_{k1} & G_{k2} & \cdots & G_{kn}{}^{1}
\end{array}
$$

The string is packed and consists of 1, 2, 4 or 8 bit characters (G_{ij}) depending on the gray level encoding.

If the string does not end on a byte boundary (G_{kn}) the remainder of the last byte is set to zero. The flag indicates the end of the scan line.

2.4.7.1.2 GBW Byte

The GBW Byte has the following format:

0	1	2	3	4	5	6	7	8

Bits 0-1	Gray scale level encoding
	00 - 1 Bit
	01 - 2 Bits
	10 - 4 Bits
	11 - 8 Bits
Bits 2-3	Border to be used
	00 - North
	01 - East
	10 - South
	11 - West
Bits 4-5	Specify width of data to be manipulated
	00 - Interior (32 bits)
	01 - Interior + Right Border (33 bits)
	10 - Interior + Left Border (33 bits)
	11 - Interior + Both Border (34 bits)

Bits 6, 7 8 are not used

2.4.7.2 Loadb

LØADB | 1 1 | 0 1 0 0 0 0 | [1] <GBW Byte><Operand>$_S$

Load border: Load border from storage according to raster string (See Section 2.4.7.1.1). The GBW Byte (See Section 2.4.7.1.2) specifies the number of bits of Gray scale and the Border to be loaded. GBW also specifies the Width of data to be loaded; either border (33 bits), both borders (34 bits), or neither border (32 bits).

The operand phrase gives the address of the raster string in core.

The previous contents of the border are overwritten.

The following table lists the planes loaded for a given gray scale.

Number of bits of gray (G)	Planes used (LSB last)
1	P0
2	P1, P0
4	P3, P2, P1, P0
8	P7, P6, P5, P4, P3, P2, P1, P0

Indicators: Parity

2.4.7.3 Storeb

STØREB [1 1 | 0 1 0 0 1 0] [1] <GBW Byte>

Store border: Store border by sending it to the ØS in raster string format (see Section 2.4.7.1.1). GBW (see Section 2.4.7.1.2) specifies the number of bits of Gray scale and the Border to be stored. GBW also specifies the Width of data to be stored; either border (33 bits), both borders (34 bits), or neither border (32 bits).

The following table lists the planes stored for a given gray scale.

Number of bits of gray (G)	Planes used (LSB last)
1	P0
2	P1, P0
4	P3, P2, P1, P0
8	P7, P6, P5, P4, P3, P2, P1, P0

Indicators: Parity

2.4.7.4 Pushb

PUSHB [1 1 | 0 1 0 0 0 1] [1] <GBW Byte><Operand>$_S$

Row load: Shift plane(s), including borders, one row in the direction away from the border being loaded. Then load border according to the raster string (see Section 2.4.7.1.1) received from storage. GBW (see Section 2.4.7.1.2) specifies the number of bits of Gray scale and the Border to be loaded. GBW also specifies the Width of data to be loaded; either border (33 bits), both borders (34 bits), or neither (32 bits).

The operand phrase gives the address of the raster string in core.

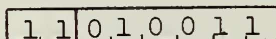
The previous contents of the border are overwritten.

The following table lists the planes loaded for a given gray scale.

Number of bits of gray (G)	Planes used (LSB last)
1	P0
2	P1, P0
4	P3, P2, P1, P0
8	P7, P6, P5, P4, P3, P2, P1, P0

Indicators: Parity

2.4.7.5 Popb

PØPB  <GBW Byte>

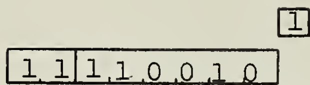
Row store: Store border by sending it to the ØS in a raster string format (see Section 2.4.7.1.1). Then shift the plane(s), including borders, one row in the direction of the border being stored. GBW (see Section 2.4.7.1.2) specifies the number of bits of Gray scale and the Border to be stored. GBW also specifies the Width of data to be stored; either border (33 bits), both borders (34 bits), or neither border (32 bits).

The following table lists the planes stored for a given gray scale.

Number of bits of gray (G)	Planes used (LSB last)
1	P0
2	P1, P0
4	P3, P2, P1, P0
8	P7, P6, P5, P4, P3, P2, P1, P0

Indicators: Parity

2.4.7.6 <u>Moveb</u>

MØVEB  <GBW Byte><Operand S_1>
<Operand S_2>

Move and copy border: Shift S_2 (in wrap-around mode) one row in direction of border specified by GBW byte (see Section 2.4.7.1.2). Then copy contents of this border of S_2 into the opposite border of S_1. Then shift S_2 back to its original position.

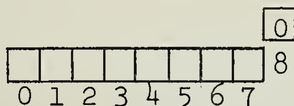
Both planes must be in the IA and neither may be the M plane.

Indicators: Parity, Invalid Plane Address.

2.4.8 Information on Boole Instruction

2.4.8.1 Availability List

The format for the availability list is:



The planes available for scratch pad uses are indicated by setting the appropriate bits to one. Note the flag bit of the availability list is always 0.

2.4.8.2 Form for Specifying General Boolean Functions

The syntax of the canonical form is formally defined as follows:

<function> ::= <term string> ; P.

<term string> ::= <term> |<term> ; <term string>

<term> ::= <elementary function> |<elementary function> <operator string>

<operator string> ::= <secondary operator> |<secondary operator>
<operator string>

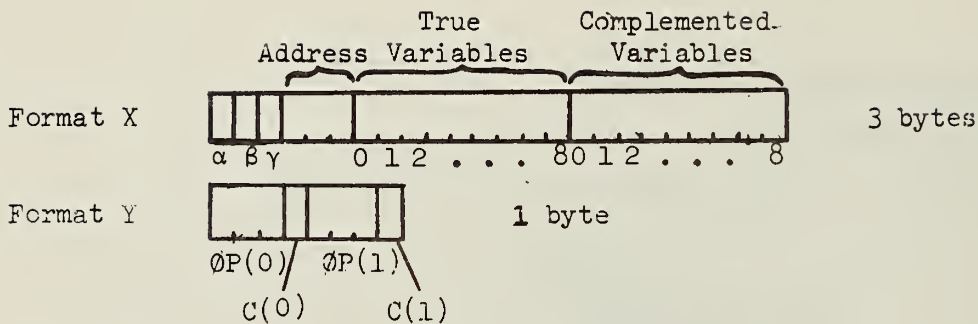
Here an elementary function is defined as any function meeting both of the following criteria:

(1) The domain of the function is restricted either to (a) any one plane, or (b) the vertical column passing through bit position 0.

(2) The function is a simple Boolean sum (OR) or a simple Boolean product (AND) of the individual (true and/or complemented) variables.

'P' identifies the output plane and '.' is an end-of-string indicator.

A secondary operator is defined as one of the conventional AND (&), OR (|), and negation (¬) of Boolean Algebra. The Polish string will be presented to the PAU in either of two formats. The two formats are:



The first information is in format X. From then on, the succeeding format is context dependent. The meanings of the individual bits and fields are now described.

Format X

α	Meaning
0	(OR) operators connect the variables of the elementary function
1	(AND) operators connect the variables of the elementary function

β	Meaning
0	a horizontal elementary function is specified
1	a vertical elementary function is specified

γ	Meaning
0	if horizontal elementary function, operand plane is as previously defined
1	if horizontal elementary function, operand plane is redefined by address field

True Variables

This is a list of all true variables appearing in the elementary function.

Complemented Variables

This is a list of all complemented variables appearing in the elementary function.

Address

If $\beta = 0$ and $\gamma = 1$, the address field defines a horizontal plane as follows (otherwise, the field is ignored):

A B C	Plane
0 0 0	0
0 0 1	1
0 1 0	2
0 1 1	3
1 0 0	4
1 0 1	5
1 1 0	6
1 1 1	7

Format Y (where s = 0, 1)

C(s)	Meaning
0	Stop (i.e., period)
1	Continue
OP(s)	If C(s) = 1, the three-bit OP(s) field has the following interpretation:

N L P	Meaning
0 0 0	|
0 0 1	|;
0 1 0	&
0 1 1	&;
1 0 0	¬
1 0 1	¬;
1 1 0	Forbidden
1 1 1	;

In other words, the individual bits have the following meanings:

N	Meaning
0	a binary secondary operator is specified by bit L
1	the operator is negation (unary), unless ";"

L	Meaning
0	if binary operator, operator is | (OR)
1	if binary operator, operator is & (AND)

P	Meaning
0	no punctuation follows operator
1	punctuation following operator is ";". NOTE: The ";" when not preceded by an operator is interpreted as meaning "next format is of type X" regardless of whether it appears in OP(1).

If C(s) = 0, the OP(s) field bits (N,L,P) are interpreted as the address of the final horizontal plane:

N L P	Plane
0 0 0	0
0 0 1	1
0 1 0	2
0 1 1	3
1 0 0	4
1 0 1	5
1 1 0	6
1 1 1	7

In actual use, the complete set of auxiliary information required to describe a function will consist of an initial format X group, followed by a mix of data in formats X and Y. Format X is used when a new elementary function is to be introduced, and format Y is used to introduce secondary operators. The address in which the result is to be stored is given in the final format Y byte.

2.4.8.3 Examples Illustrating the Coding

2.4.8.3.1 Purely Horizontal Logic

Consider the function

P6 = ¬(¬(10)&(12)&(16)|(20))|(34)&(38)

In the canonical form it is written (where 'F' is the name of the Boolean function)

F: [¬(10)&(12)&(16)];[20]|¬;[(34)&(38)]|;6.

The code for this string is (Figure 2.4.8.3.1/1).

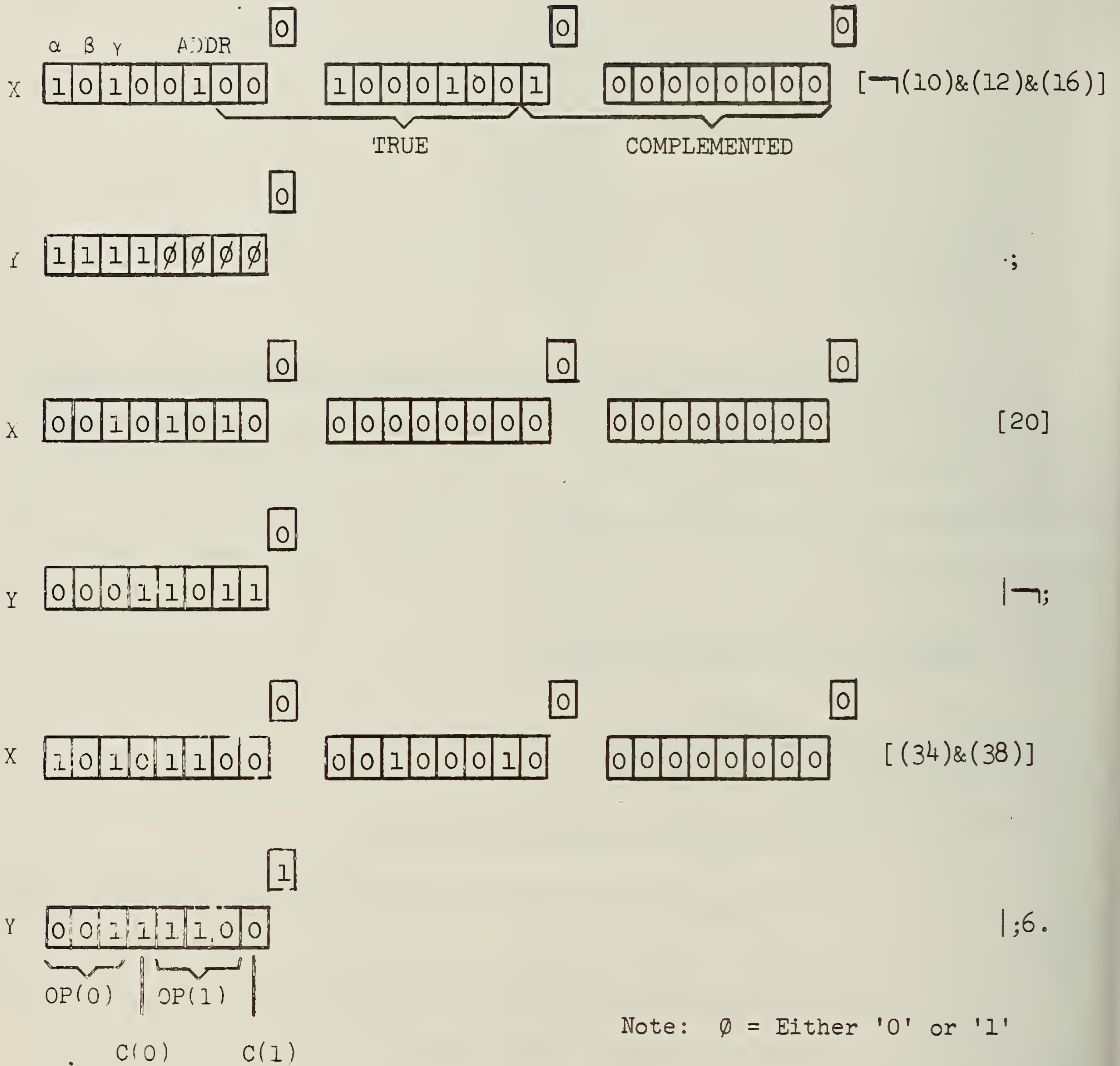


Figure 2.4.8.3.1/1 Code for Horizontal Logic Example

2.4.8.3.2 Purely Vertical Logic

Consider the example

P5 = (20)&(30)|¬(60)

In the canonical form, it is written (where 'F' is the name of the Boolean function)

F:[(20)&(30)];[¬(60)]|;5.

The code for this string is (Figure 2.4.8.3.2/1).

Figure 2.4.8.3.2/1 Code for Vertical Logic Example

2.4.8.3.3 General Three-Dimensional Logic

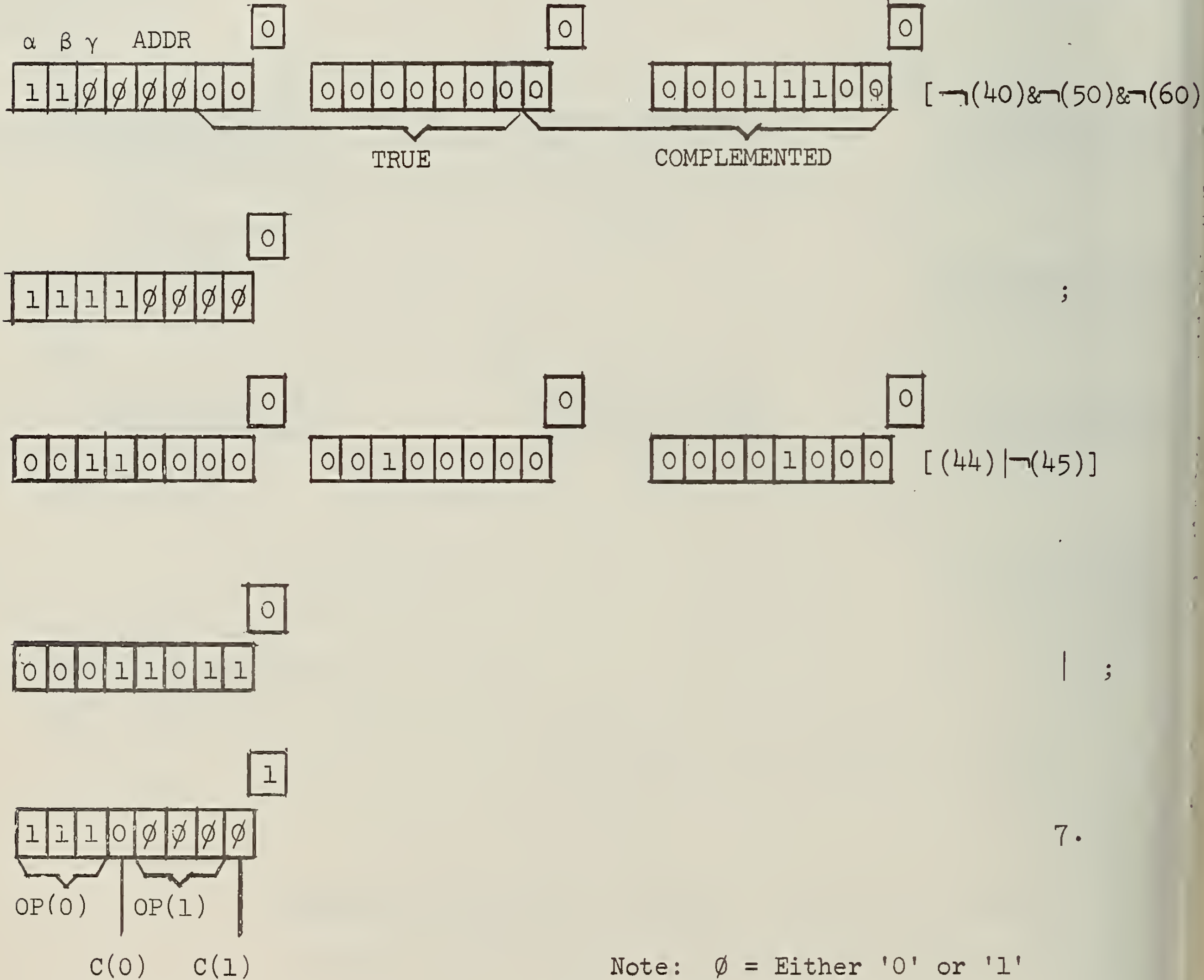
Consider the function

$$P7 = \neg(\neg(40) \& \neg(50) \& \neg(60) \mid (44) \mid \neg(45))$$

A canonical equivalent is (where 'F' is the name of the Boolean function)

$$F:[\neg(40) \& \neg(50) \& \neg(60)] \; ;[(44) \mid \neg(45)] \mid \neg;7.$$

The code for this canonical form is (Figure 2.4.8.3.3/1).



Note: ∅ = Either '0' or '1'

Figure 2.4.8.3.3/1 Code for Example of General Three-Dimensional Logic Having Both Horizontal and Vertical Components

AEC-427
(6/68)
CM 3201

U.S. ATOMIC ENERGY COMMISSION
UNIVERSITY-TYPE CONTRACTOR'S RECOMMENDATION FOR DISPOSITION OF SCIENTIFIC AND TECHNICAL DOCUMENT

(See Instructions on Reverse Side)

AEC REPORT NO. 434 COO-2118-0006	2. TITLE ILLIAC III REFERENCE MANUAL VOLUME II: Instruction Repertoire

3. TYPE OF DOCUMENT (Check one):

[X] a. Scientific and technical report
[ ] b. Conference paper not to be published in a journal:
Title of conference ______
Date of conference ______
Exact location of conference ______
Sponsoring organization ______
[ ] c. Other (Specify) ______

4. RECOMMENDED ANNOUNCEMENT AND DISTRIBUTION (Check one):

[X] a. AEC's normal announcement and distribution procedures may be followed.
[ ] b. Make available only within AEC and to AEC contractors and other U.S. Government agencies and their contractors.
[ ] c. Make no announcement or distrubution.

5. REASON FOR RECOMMENDED RESTRICTIONS:

6. SUBMITTED BY: NAME AND POSITION (Please print or type)

B. H. McCormick
Principal Investigator
Illiac III Project

Organization Department of Computer Science
University of Illinois
Urbana, Illinois

Signature Bruce H. McCormick	Date February 26, 1971

FOR AEC USE ONLY

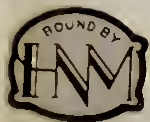
AEC CONTRACT ADMINISTRATOR'S COMMENTS, IF ANY, ON ABOVE ANNOUNCEMENT AND DISTRIBUTION RECOMMENDATION:

--

PATENT CLEARANCE:

[ ] a. AEC patent clearance has been granted by responsible AEC patent group.
[ ] b. Report has been sent to responsible AEC patent group for clearance.
[ ] c. Patent clearance not required.

2 7 1979


BOUND BY
HNM

UNIVERSITY OF ILLINOIS-URBANA
510.84 IL6R no. C002 no.433-438(1971
Internal report /